Fundamentos de los Sistemas Telemáticos

Tema 2: Representación de la información

Gregorio Fernández Fernández

Departamento de Ingeniería de Sistemas Telemáticos
Escuela Técnica Superior de Ingenieros de Telecomunicación
Universidad Politécnica de Madrid

Este documento está diseñado para servir de material de estudio a los alumnos de la asignatura «Fundamentos de los Sistemas Telemáticos» de la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universidad Politécnica de Madrid.

La licencia cc-by-sa significa que usted es libre de copiar, de distribuir, de hacer obras derivadas y de hacer un uso comercial del documento, con dos condiciones: reconocer el origen citando los datos de la portada, y utilizar la misma licencia.

© DIT-UPM, 2012. Algunos derechos reservados.
Este material se distribuye con la licencia Creative Commons «by–sa» disponible en: http://creativecommons.org/licenses/by-sa/3.0/deed.es

Índice general

1 Terminología y objetivos **1**
- 1.1 Datos, información y conocimiento 1
- 1.2 Caudal y volumen 3
- 1.3 Soportes de transmisión y almacenamiento 4
- 1.4 Extremistas mayores y menores 6
- 1.5 Objetivos y contenido del Tema 8

2 Representación de datos textuales **9**
- 2.1 Codificación binaria 9
- 2.2 Código ASCII y extensiones 10
- 2.3 Códigos ISO 11
- 2.4 Unicode 12
- 2.5 Cadenas 14

3 Representación de datos numéricos **15**
- 3.1 Números enteros sin signo 15
- 3.2 Formatos de coma fija 16
- 3.3 Números enteros 16
- 3.4 Operaciones básicas de procesamiento 17
- 3.5 Números racionales 22

4 Detección de errores y compresión **27**
- 4.1 Bit de paridad 28
- 4.2 Suma de comprobación *(checksum)* 29
- 4.3 Control de errores 30
- 4.4 Compresión sin pérdidas *(lossless)* 31
- 4.5 Compresión con pérdidas *(lossy)* 34

5 Almacenamiento en ficheros **39**
- 5.1 Tipos de ficheros 39
- 5.2 Ficheros regulares 40
- 5.3 Ficheros de texto, archivos, documentos y datos estructurados 43
- 5.4 Ficheros multimedia 45
- 5.5 Ficheros ejecutables binarios 53

Apéndice: Sistemas de numeración posicionales **55**

Bibliografía **59**

Capítulo 1

Terminología y objetivos

En este capítulo introductorio se presentan algunos conceptos generales y se concretan los objetivos del Tema.

1.1. Datos, información y conocimiento

Con frecuencia utilizamos los términos «datos» e «información» como sinónimos, pero conviene diferenciarlos, y las definiciones «oficiales» no ayudan mucho. Las dos acepciones de «información» del Diccionario de la Real Academia Española más relevantes para nuestro propósito son:

> *5. f. Comunicación o adquisición de conocimientos que permiten ampliar o precisar los que se poseen sobre una materia determinada.*
>
> *6. f. Conocimientos así comunicados o adquiridos.*

La primera remite a los mecanismos por los que se adquieren los conocimientos, y la segunda parece identificar «información» con «conocimientos», concepto que define en singular:

> *Conocimiento:*
>
> *2. m. Entendimiento, inteligencia, razón natural.*

Y «dato» es:

> *1. m. Información sobre algo concreto que permite su conocimiento exacto o sirve para deducir las consecuencias derivadas de un hecho.*
>
> *3. m. Inform. Información dispuesta de manera adecuada para su tratamiento por un ordenador.*

¿Es la información conocimiento? ¿Son los datos información?

Para buscar definiciones más prácticas, operativas y orientadas a la ingeniería, podemos basarnos en un concepto de la rama de la Informática llamada «inteligencia artificial»: el concepto de *agente*[1]:

Un **agente** es un sistema que percibe el entorno y puede actuar sobre él. Se supone que el agente es *racional*: tiene conocimientos sobre un dominio de competencia, mecanismos de razonamiento y

[1] Una explicación más detallada, con fuentes bibliográficas, de esta definición y las siguientes puede encontrarse en el documento «Representación del conocimiento en sistemas inteligentes», `http://www.gsi.dit.upm.es/~gfer/ssii/rcsi/`.

objetivos, y utiliza los dos primeros para, en función de los datos que percibe, generar las acciones que conducen a los objetivos.

Debe usted interpretar «entorno», «percepción» y «actuación» en un sentido general: el agente puede ser una persona o un robot con sensores y actuadores sobre un entorno físico, pero también puede ser un programa que, por ejemplo, «percibe» el tráfico de paquetes de datos en una red (su entorno), los analiza y «actúa» avisando a otros agentes de posibles intrusiones.

En este contexto, entender las diferencias entre «datos», «información» y «conocimiento» es fácil:

- Los **datos** son estímulos del entorno que el agente percibe, es decir, meras entradas al sistema.
- La **información** está formada por datos *con significado*. El significado lo pone el agente receptor de los datos, que, aplicando sus mecanismos de razonamiento, los *interpreta* de acuerdo con sus conocimientos previos.
- El **conocimiento** es la información una vez asimilada por el agente en una forma sobre la que puede aplicar razonamientos.

Según estas definiciones, no tiene sentido decir que un libro o un periódico o la web, por ejemplo, contienen conocimiento, ni siquiera información. Solo podría ser así si el libro, el periódico o la web tuviesen capacidad de razonamiento. Lo que contienen son datos, y lo que sí puede decirse de ellos es que son fuentes de información y conocimiento para los agentes capaces de asimilar esos datos. Seguramente ha oido usted hablar de la «web semántica»: los datos de las páginas, interpretables por agentes humanos, se complementan con más datos expresados en lenguajes interpretables por agentes artificiales. El nombre es un poco falaz, porque sin agentes no hay semántica.

La definición de información como «datos con significado para un agente» es compatible con la que estudiará usted en una asignatura de tercer curso llamada «Teoría de la información»: la cantidad de información de un mensaje se mide por la reducción de incertidumbre en el agente receptor, es decir, depende del conocimiento previo de ese receptor. Si el periódico me dice que el Real Madrid ha ganado al Barça, la cantidad de información que me comunica es 1 bit, pero si yo ya lo sabía es 0 bits.

¿Qué es un bit?

Es posible que el final del párrafo anterior le haya sorprendido. Porque usted sabe interpretar el significado de «bit» en otros contextos. Por ejemplo, cuando una llamada telefónica irrumpe en su intimidad y una edulcorada voz le propone un servicio de «banda ancha» de «x megas». Usted entiende que «megas» quiere decir «millones de bits por segundo», y que lo que le está ofreciendo el inoportuno vendedor (o vendedora) es un *caudal de datos*, no de información. Si compra un *pecé* con un disco de un *terabyte* usted sabe que su *capacidad de almacenamiento de datos* es, aproximadamente, un billón de bytes, y que cada byte son ocho bits. Que ese caudal de datos o que los datos guardados en el disco contengan o no información para usted es otro asunto.

Ocurre que con el término «bit» designamos dos cosas que guardan cierta relación pero son diferentes:

- «Bit» es la unidad de medida de información: es la información que aporta un mensaje a un agente cuando este mensaje le resuelve una duda entre dos situaciones igualmente probables. En la asignatura «Teoría de la información» verá usted cómo se define con rigor en términos matemáticos.
- «Bit» es la contracción de «binary digit»: es un *dígito binario* con dos valores posibles, «0» o «1». Éste es el significado que aplicaremos en adelante en este Tema.

Nivel de abstracción lógico

Todos los datos que manejamos en Telemática están *codificados en binario*, es decir, un dato está siempre formado por bits. Físicamente, un bit se materializa mediante algún dispositivo que puede estar en uno de dos estados posibles. Por ejemplo un punto en un circuito eléctrico que puede tener +5 voltios o 0 voltios, o una pequeña zona en la superficie de un disco magnetizada en un sentido o el opuesto. Nosotros hacemos abstracción de esos detalles físicos: todo está basado en componentes que pueden estar en uno de dos estados, y llamamos a esos estados «0» y «1».

Resumiendo...

Lo que vamos a estudiar en este Tema son convenios para representar, almacenar y transmitir distintos *tipos de datos:* textos, números, sonidos, imágenes, etc. Los datos forman parte de mensajes que envía un agente emisor a otro receptor a través de un *canal de comunicación*, y es necesario que ambos se atengan a los mismos convenios para que puedan entenderse. En un acto de comunicación, cada dato que llega al receptor puede aportarle información o no, dependiendo del estado de sus conocimientos, pero éste es un asunto en el que no entraremos.

1.2. Caudal y volumen

El **caudal** de un canal de comunicación es la **tasa de bits** *(bitrate)*, medida en bits por segundo. Se suele abreviar como «bps», y se utilizan los múltiplos habituales del Sistema Internacional: kilo, mega, etc.

Para un caudal constante, es trivial calcular el **volumen** de datos transmitido en un intervalo de tiempo mediante una multiplicación. Si, cediendo ante la insistencia de su convincente vendedor, contrata un servicio de 50 «megas» (50 Mbps), usted razonablemente espera que en una hora de conexión recibirá $50 \times 3.600 \times 10^6 = 180$ gigabits. Esto es correcto (si se cumple la promesa del vendedor), pero hay que matizar dos cosas, una que seguramente le es familiar, la otra quizás no tanto.

Sin duda le resulta familiar la palabra «byte». El volumen de datos normalmente está relacionado con la **capacidad de almacenamiento** de algún sistema capaz de guardarlos, lo que solemos llamar «memoria». En estos sistemas la unidad básica de almacenamiento es un conjunto de ocho bits, conocido como **octeto**, o **byte**, y ésta es también la unidad para expresar la capacidad. De modo que si usted pretende guardar en el disco de su ordenador los datos recibidos en el ejemplo anterior necesitará disponer de al menos $180/8 = 22,5$ GB (gigabytes). En ocasiones conviene referirse a la mitad de un byte, cuatro bits: en inglés se llama *«nibble»*, que podemos traducir por **cuarteto**.

kilo y kibi, mega y mebi...

La segunda cosa a matizar es lo que queremos decir al hablar de una memoria de «x gigabytes». Más adelante veremos que la capacidad de almacenamiento de una memoria, medida en número de bytes, es siempre una potencia de 2. No tiene sentido una memoria de exactamente 1 KB. La potencia de 2 más próxima a 1.000 es $2^{10} = 1.024$. Coloquialmente solemos decir «kilobytes», «megabytes», etc., pero hablando de capacidad de almacenamiento de una memoria es incorrecto: los prefijos multiplicadores definidos en el Sistema Internacional de Medidas son potencias de diez, no de dos. Hay un estándar definido por la IEC (International Electrotechnical Commission) para expresar los multiplicadores binarios: «kibi» (kilo binary), «mebi» (mega binary), «gibi» (giga binary), etc. Es el resumido en la tabla 1.1, y el que utilizaremos en lo sucesivo.

Decimales (SI)		Binarios (IEC)	
Valor	**Prefijo**	**Valor**	**Prefijo**
10^3	kilo (k)	2^{10}	kibi (Ki)
10^6	mega (M)	2^{20}	mebi (Mi)
10^9	giga (G)	2^{30}	gibi (Gi)
10^{12}	tera (T)	2^{40}	tebi (Ti)
10^{15}	peta (P)	2^{50}	pebi (Pi)
10^{18}	exa (E)	2^{60}	exbi (Ei)
10^{21}	zetta (Z)	2^{70}	zebi (Zi)
10^{24}	yotta (Y)	2^{80}	yobi (Yi)

Tabla 1.1 Prefijos multiplicadores.

Binario y hexadecimal

Hex.	Bin.
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Se supone que entre sus conocimientos previos están los que se refieren a los sistemas de numeración. Si no es así, acuda al apéndice de este documento en el que se resumen.

Todos los datos se codifican en binario para que puedan ser tratados por los sistemas digitales. Pero «hablar en bits», como hacen ellos (los sistemas), es prácticamente imposible para la mente humana. Sin embargo, a veces tenemos que referirnos al valor concreto de un byte, o de un conjunto de bytes. Por ejemplo, en el siguiente capítulo veremos que el carácter «Z» se codifica así: 01011010. Es casi inevitable equivocarse al escribirlo, y mucho más si hablamos de una sucesión de caracteres o, en general, de un contenido binario de varios bytes.

Una posibilidad es interpretar «01011010» como la expresión en base 2 de un número entero: $2^6+2^4+2^3+2^1 = 90$, y decir que la codificación de «Z» es 90. Pero más fácil es utilizar el sistema hexadecimal (base 16), porque la conversión de hexadecimal a binario o a la inversa es inmediata: basta con agrupar los bits en cuartetos y tener en cuenta la correspondencia entre dígitos hexadecimales y su expresión en binario.

La codificación «binaria» (expresada en hexadecimal) de «Z» es: **0x5A**. «0x» es el prefijo más común para indicar que lo que sigue está expresado en hexadecimal.

1.3. Soportes de transmisión y almacenamiento

Para la transmisión de datos existe una variedad de medios físicos: cable de cobre trenzado, coaxial, fibra óptica, medios no guiados, etc., con distintas características de ancho de banda, coste, etc. No vamos a decir nada más de ellos aquí, porque se estudian en la asignatura «Introducción a la ingeniería». Pero sí tenemos que dar algunos detalles sobre los soportes de almacenamiento.

Un **punto de memoria** es un mecanismo físico capaz de almacenar un bit. Hay diversas tecnologías (electrónicas, magnéticas, ópticas, etc.) que permiten implementar este mecanismo, dando lugar a diferentes **soportes** para almacenar bits. Los soportes tienen características variadas:

- **capacidad**: número de bits, medido en bytes, o KiB, etc.;

- posibilidad de **escritura** (grabación de un conjunto de bits) o de solo **lectura** (recuperación de un conjunto de bits);
- **tiempos de acceso** para la lectura y la escritura;
- **coste por bit**;
- **densidad de integración**: número de bits por unidad de superficie;
- **volatilidad**: el soporte es volátil si requiere alimentación eléctrica para conservar los contenidos almacenados.

Generalmente el subsistema de memoria de un sistema informático es una combinación de varios soportes, lo que conduce al concepto de **jerarquía de memorias**. Lo único que a los efectos de este Tema nos interesa es conocer algunas características funcionales básicas de esos soportes para comprender los convenios de almacenamiento de los datos.

Biestables y registros

Un **biestable**, o **flip-flop**, es un circuito que estudiará usted en la asignatura «Electrónica digital». En todo momento, el biestable se encuentra en uno de dos estados posibles y permanece en él mientras no se le aplique un estímulo para cambiar de estado. Por tanto, sirve como punto de memoria. Veremos en el capítulo 3 que, acompañando al resultado de las operaciones aritméticas y lógicas, hay **indicadores** que toman uno de entre dos valores indicando si ese resultado fue negativo, si se ha producido desbordamiento, etc. Esos indicadores son biestables.

En el Tema 3 estudiaremos los procesadores. Un procesador hardware contiene **registros**, que no son más que conjuntos de biestables. En cada registro puede almacenarse un conjunto de bits conocido como «palabra». La **longitud de palabra** es el número de bits que contienen los registros. Valores típicos son 8, 16, 32 o 64 (uno, dos, cuatro u ocho bytes), dependiendo del procesador.

Los registros son volátiles, pero son los soportes más rápidos dentro de la jerarquía de memorias. Los tiempos de acceso para la escritura (grabación de una palabra) y para la lectura (recuperación del contenido) son de pocos nanosegundos. Y los registros son también los soportes que tienen mayor coste por bit y menor densidad de integración. Los procesadores pueden incluir varias decenas de registros. Con 30 registros de una longitud de palabra de 64 bits resulta una capacidad de $30 \times 64/8 = 240$ bytes. Una capacidad del orden de KiB solo con registros es, hoy por hoy, impracticable, tanto por su coste como por su tamaño.

Memorias de acceso aleatorio

La memoria principal y las memorias ocultas (más conocidas como *«caches»*) se implementan mediante otras tecnologías electrónicas con las que se consiguen costes por bit y densidades de integración tales que actualmente es factible llegar a capacidades de almacenamiento del orden de KiB y MiB (para las memorias ocultas) y de GiB (para la memoria principal).

La memoria principal y las ocultas se diferencian en los tiempos de acceso (mayores los de la memoria principal), en las capacidades de almacenamiento (también mayores en la memoria principal, debido a la mayor densidad de integración de la tecnología empleada) y en los costes por bit (menores en la memoria principal). Por lo demás, ambas son volátiles, ambas son de lectura y escritura y en ambas se puede acceder a un byte o a varios bytes (normalmente a una palabra) en una sola operación.

Para poder acceder a un byte determinado es necesario poder identificarlo. Cada byte tiene asignada una **dirección**, que es un número entero comprendido entre 0 y $M - 1$, donde M es la **capacidad** de la memoria. Cuando tiene que leer (extraer) o escribir (grabar) un byte, el procesador genera su dirección y la pone en un registro para que los circuitos de la memoria para activen el byte (o la palabra) que corresponda. Si el registro tiene n bits, las direcciones que puede contener están comprendidas entre 0 y 2^{n-1}. Por ejemplo, con un registro de 16 bits las direcciones posibles son 0x0000, 0x0001... 0xFFFF: en total, $2^{16} = 65.536$ direcciones. En este caso, la máxima capacidad de memoria que puede direccionarse es $2^{16} = 2^6 \times 2^{10}$ bytes = 64 KiB. Con 32 bits resulta $2^{32} = 4$ GiB.

«Acceso aleatorio» no significa «acceso al azar». Solo quiere decir que el tiempo de acceso para la lectura o la escritura es el mismo para todos los bytes, independiente de la dirección. Estas memorias se conocen habitualmente como «**RAM**» (Random Access Memory)[2]. Conviene aclarar un malentendido bastante común. Es frecuente que una pequeña parte de la memoria está construida con una tecnología que no es volátil, para conservar los programas de arranque y algunas rutinas básicas. Esta tecnología permite construir una memoria no volátil, pero de solo lectura, o «**ROM**» (Read Only Memory): una vez grabada por el fabricante no puede borrarse ni volver a escribirse[3]. Esto ha conducido a que en la jerga habitual se distinga entre RAM y ROM como si fuesen cosas excluyentes, cuando la ROM es también RAM. Sí son excluyentes «R/W» («Read/Write»: lectura/escritura) y «ROM».

Memorias secundarias y terciarias

Diversas tecnologías (discos y cintas magnéticos, discos ópticos, *flash*, etc.) permiten construir soportes para el almacenamiento masivo de datos fuera de la memoria principal. De la gestión de esta **memoria secundaria** se ocupa automáticamente el sistema operativo, mientras que la **memoria terciaria** es aquella que requiere alguna intervención humana (como insertar o extraer un DVD).

En general, estos soportes se diferencian de los utilizados para implementar la memoria principal en que:

- no son volátiles,
- el acceso a los datos no es «aleatorio», y el tiempo medio de acceso es mayor, y
- no se comunican directamente con el procesador: los datos se leen o se escriben en la memoria principal en **bloques** de un tamaño mínimo de 512 bytes, mediante la técnica de acceso directo a memoria (DMA).

El convenio de almacenamiento en las memorias secundarias y terciarias se basa en estructurar los datos en **ficheros** (también llamados «**archivos**», ver apartado 4.4) como los que hemos manejado en el Tema 1. En el capítulo 5 veremos algunos convenios concretos para el almacenamiento en ficheros.

1.4. Extremistas mayores y menores

Antes de entrar a estudiar los convenios de representación de los distintos tipos de datos, hay un convenio general sobre el almacenamiendo en las memorias de acceso aleatorio que debemos conocer.

Hemos dicho que cada dirección identifica a un byte dentro de la memoria principal (o la oculta). Y también que en la memoria se pueden almacenar palabras que, normalmente, ocupan varios bytes: dos, cuatro u ocho. ¿Cómo se identifica a una palabra, para poder escribirla o leerla? Por la dirección

[2] Aunque se sigue utilizando el nombre «RAM», las actuales «DRAM» (RAM dinámicas) no son, en rigor, de acceso aleatorio, porque en ellas no se accede a los datos byte a byte, o palabra a palabra, sino a ráfagas.

[3] En realidad, se usan tecnologías que no son estrictamente ROM, porque la memoria puede regrabarse, aunque a menor velocidad. Se llaman EEPROM (Electrically Erasable Programmable ROM).

de uno de sus bytes, pero ¿en qué direcciones se almacenan éstos? Parece obvio que en direcciones consecutivas, pero ¿en qué orden?

Gráficamente se entenderán mejor los posibles convenios para responder a esas preguntas. La representación gráfica de una palabra de n bits es un rectángulo horizontal con una numeración de 0 a $n-1$ de los bits que componen la palabra, como muestra la figura 1.1 para el caso de $n = 16$. Esta numeración coincide con los pesos de los bits cuando el conjunto se interpreta como un número entero en binario. Al bit de peso 0 le llamamos «bit menos significativo» («bms» en la figura) y al de peso $n-1$ «bit más significativo» («bMs»).

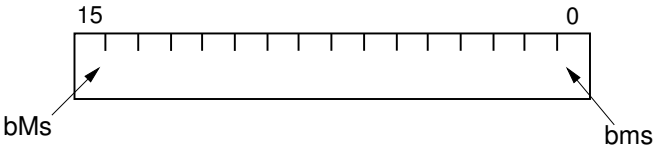

Figura 1.1. Representación gráfica de una palabra de 16 bits.

Por otra parte, para la memoria utilizaremos gráficos como el de la figura 1.2, con las direcciones en el margen izquierdo y empezando desde arriba con la dirección 0. Esta figura correspondería a una memoria de 16 MiB. En efecto, las direcciones tienen seis dígitos hexadecimales, y la dirección más alta (que está en la parte más baja de la figura) es 0xFFFFFF $= 16^6 - 1$. El número total de direcciones es: $16^6 = 2^4 \times 2^{20}$.

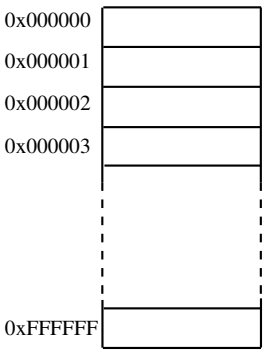

Figura 1.2. Representación gráfica de una RAM.

Volviendo a las preguntas anteriores sobre el almacenamiento de palabras, y suponiendo palabras de 16 bits, la figura 1.3 muestra los dos posibles convenios para almacenar sus dos bytes en las direcciones d y $d+1$. El convenio de que al byte que contiene los bits menos significativos (del 0 al 7) le corresponda la dirección menor se le llama **extremista menor** *(little endian)*, y al opuesto, **extremista mayor** *(big endian)*.

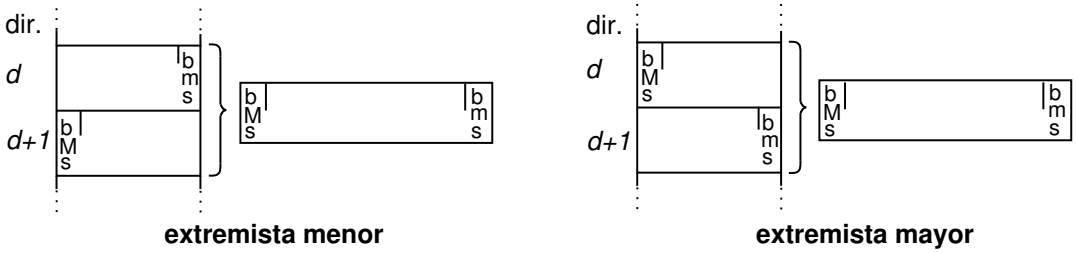

Figura 1.3 Convenios de almacenamiento para una palabra de dos bytes.

La generalización para otras longitudes de palabra es bastante obvia. Así, una palabra de 32 bits ocupará cuatro direcciones, desde la d a la $d+3$. Con el convenio extremista menor al byte menos significativo le corresponde la dirección d, mientras que con el extremista mayor le corresponde la $d+3$. En ambos casos, se considera que la dirección de la palabra es la más baja de las cuatro, d.

La diferencia de convenios afecta no solo al almacenamiento, también al orden en que se envían los bytes en las comunicaciones. Algunos procesadores (por ejemplo, los de Intel) están diseñados con el convenio extremista menor, y otros (por ejemplo, los de Motorola) con el extremista mayor. El problema surge cuando uno de ellos interpreta una sucesión de bytes enviada por el otro. Por ejemplo, las direcciones IP (versión 4) se representan como un conjunto de 32 bits: cada una de las cuatro partes separadas por puntos es la traducción a decimal de un byte. Así, «138.4.2.61» en realidad es «0x8A04023D». Si una máquina con procesador Motorola genera esta dirección, envía los bytes por la red en el orden 8A-04-02-3D (primero lo más significativo) y lo recibe otra con procesador Intel, para

ésta lo que llega primero es lo menos significativo, resultando que interpreta la dirección «61.2.4.138». Por cierto, los protocolos de la familia TCP/IP son extremistas mayores, por lo que es en el software de la máquina basada en Intel en el que hay que invertir el orden para que la comunicación sea correcta.

1.5. Objetivos y contenido del Tema

La Guía de aprendizaje de la asignatura establece que los resultados del aprendizaje relacionados con este Tema son:

- Conocer los convenios de representación binaria, transmisión y almacenamiento de la información.
- Conocer los principios de los algoritmos de detección de errores y compresión

Estudiaremos los convenios de representación en bajo nivel (en binario) de datos de tipo textual (capítulo 2) y numérico (capítulo 3), así como los principios de la detección de errores y la compresión (capítulo 4). Los datos de tipo audiovisual y los que proceden de sensores del entorno físico requieren para su representación en binario procesos de digitalización que no explicaremos aquí porque forman parte del programa de la asignatura «Introducción a la ingeniería». Pero sí veremos los fundamentos de los algoritmos para su compresión (capítulo 4) y algunos convenios para su almacenamiento en ficheros (capítulo 5).

Capítulo 2

Representación de datos textuales

Si vamos a tratar de «representación de textos» hay que empezar precisando qué es lo que queremos representar. Lo más sencillo es el «texto plano»: el formado por secuencias de **grafemas** (a los que, más familiarmente, llamamos «caracteres»), haciendo abstracción de su tamaño, color, tipo de letra, etcétera (que dan lugar a los **alógrafos** del grafema). Los grafemas son los caracteres visibles, pero el texto plano incluye también caracteres que no tienen una expresión gráfica y son importantes: espacio, tabulador, nueva línea, etcétera.

En este capítulo nos limitamos al texto plano. Para representar los distintos alógrafos los programas de procesamiento de textos utilizan convenios propios y muy variados (apartado 5.3), pero para el texto plano veremos que hay un número reducido de estándares.

2.1. Codificación binaria

La **codificación** es la traducción de un mensaje expresado en un determinado alfabeto, $\mathcal{A}$, al mismo mensaje pero expresado en otro alfabeto, $\mathcal{B}$. La traducción inversa se llama **descodificación** (o **decodificación**). Si los mensajes han de representarse en binario, entonces $\mathcal{B} = \{0,1\}$.

La codificación se realiza mediante un **código**, o correspondencia biunívoca entre los símbolos de los dos alfabetos. Como normalmente tienen diferente número de símbolos, a algunos símbolos de un alfabeto hay que hacerles corresponder una secuencia de símbolos del otro. Por ejemplo, si $\mathcal{A} = \{a,b,c\}$ y $\mathcal{B} = \{0,1\}$, podríamos definir el código:

$\mathcal{A}$	$\mathcal{B}$
a	0
b	1
c	01

Ahora bien, tal código no sería útil, porque la descodificación es ambigua. Así, el mensaje codificado «011» puede corresponder a los mensajes originales «abb» o «cb». Si «a» se codifica como «00» el código ya no es ambiguo. Pero lo más sencillo es que el código sea de longitud fija, es decir, que el número de símbolos de $\mathcal{B}$ que corresponden a cada símbolo de $\mathcal{A}$ sea siempre el mismo. Podríamos establecer el código:

$\mathcal{A}$	$\mathcal{B}$
a	00
b	01
c	11

o bien:

$\mathcal{A}$	$\mathcal{B}$
a	01
b	00
c	10

o bien...

Los códigos BCD, ASCII y las normas ISO que veremos a continuación son de longitud fija. Pero UTF-8 (apartado 2.4) es de longitud variable. También veremos un ejemplo de codificación con longitud variable relacionado con la compresión en el apartado 4.4.

Códigos BCD

Codificar en binario un conjunto de m símbolos con un código de longitud fija requiere un número n de bits tal que $2^n \geq m$. Por tanto, para codificar los diez dígitos decimales hacen falta cuatro bits. Estos códigos se llaman **BCD** (decimal codificado en binario). El más común es el BCD «natural» (o BCD, a secas), en el que cada dígito decimal se representa por su equivalente en el sistema de numeración binario.

	BCD natural	**Aiken**	**exceso de 3**
0	0000	0000	0011
1	0001	0001	0100
2	0010	0010	0101
3	0011	0011	0110
4	0100	0100	0111
5	0101	1011	1000
6	0110	1100	1001
7	0111	1101	1010
8	1000	1110	1011
9	1001	1111	1100

Tabla 2.1 Tres códigos BCD.

Como cuatro bits permiten codificar dieciséis símbolos de los que solo usamos diez, hay $V_{16,10} = 16!/6! \approx 2{,}9 \times 10^{10}$ códigos BCD diferentes. En la Tabla 2.1 se muestran tres: el BCD «natural», u «8-4-2-1» (los números indican los pesos correspondientes a cada posición, es decir, que, por ejemplo, $1001 = 1\times8+0\times4+0\times1+1\times1 = 9$) y otros dos que presentan algunas ventajas para simplificar los algoritmos o los circuitos aritméticos: el código de Aiken, que es un código «2-4-2-1», y el llamado «exceso de 3».

2.2. Código ASCII y extensiones

En nuestra cultura se utilizan alrededor de cien grafemas para la comunicación escrita contando dígitos, letras y otros símbolos: ?, !, @, $, etc. Para su codificación binaria son necesarios al menos siete bits ($2^6 = 64; 2^7 = 128$). Para incluir letras con tilde y otros símbolos es preciso ampliar el código a ocho bits ($2^8 = 256$).

El **ASCII** (American Standard Code for Information Interchange) es una **norma**, o **estándar**, para representación de caracteres con *siete bits* que data de los años 1960 (ISO/IEC 646) y que hoy está universalmente adoptada. Las treinta y dos primeras codificaciones y la última no representan grafemas sino acciones de control. Por tanto, quedan $2^7 - 33 = 95$ configuraciones binarias con las que se codifican los dígitos decimales, las letras mayúsculas y minúsculas y algunos caracteres comunes (espacio en blanco, «.», «+», «–», «<», etc.), pero no letras con tilde, ni otros símbolos como «¿», «¡», «ñ», «ç», «€», etc.

La tabla 2.2 muestra las codificaciones ASCII a partir de 0x20 expresadas en hexadecimal y en decimal. El decimal es útil por la forma que hay en algunos sistemas para introducir caracteres que no están en el teclado: Alt+<valor decimal>.

Algunos de los caracteres de control son:

0x00, NUL: carácter nulo (fin de cadena)

0x08, BS (backspace): retroceso

0x0A, LF (line feed): nueva línea

0x0D, CR (carriage return): retorno

0x1B, ESC (escape)

0x7F, DEL (delete): borrar

Hex.	Dec.		Hex.	Dec.		Hex.	Dec.		Hex.	Dec.		Hex.	Dec.		Hex.	Dec.	
20	032		30	048	0	40	064	@	50	080	P	60	096	‘	70	112	p
21	033	!	31	049	1	41	065	A	51	081	Q	61	097	a	71	113	q
22	034	"	32	050	2	42	066	B	52	082	R	62	098	b	72	114	r
23	035	#	33	051	3	43	067	C	53	083	S	63	099	c	73	115	s
24	036	$	34	052	4	44	068	D	54	084	T	64	100	d	74	116	t
25	037	%	35	053	5	45	069	E	55	085	U	65	101	e	75	117	u
26	038	&	36	054	6	46	070	F	56	086	V	66	102	f	76	118	v
27	039	’	37	055	7	47	071	G	57	087	W	67	103	g	77	119	w
28	040	(	38	056	8	48	072	H	58	088	X	68	104	h	78	120	x
29	041	)	39	057	9	49	073	I	59	089	Y	69	105	i	79	121	y
2A	042	*	3A	058	:	4A	074	J	5A	090	Z	6A	106	j	7A	122	z
2B	043	+	3B	059	;	4B	075	K	5B	091	[	6B	107	k	7B	123	{
2C	044	,	3C	060	<	4C	076	L	5C	092	\	6C	108	l	7C	124	|
2D	045	-	3D	061	=	4D	077	M	5D	093	]	6D	109	m	7D	125	}
2E	046	.	3E	062	>	4E	078	N	5E	094	^	6E	110	n	7E	126	~
2F	047	/	3F	063	?	4F	079	O	5F	095	_	6F	111	o	7F	127	DEL

Tabla 2.2 Código ASCII.

Extensiones

ASCII es «el estándar» de 7 bits, pero no es el único. Otro, también de 7 bits pero con cambios en los grafemas representados (introduce algunas letras con tilde), es el GSM 03.38, estándar para SMS en telefonía móvil.

La mayoría de las extensiones se han hecho añadiendo un bit (es decir, utilizando un byte para cada carácter), lo que permite ciento veintiocho nuevas configuraciones, conservando las codificaciones ASCII cuando este bit es 0. Se han desarrollado literalmente miles de códigos. Si no se lo cree usted, pruebe el programa `iconv`, que traduce un texto de un código a otro. Escribiendo «`iconv -l`» se obtiene una lista de todos los códigos que conoce.

Un contemporáneo de ASCII, de ocho bits pero no compatible, que aún se utiliza en algunos grandes ordenadores *(mainframes)* es el EBCDIC (Extended Binary Coded Decimal Interchange Code).

Las extensiones a ocho bits más utilizadas son las definidas por las normas ISO 8859.

2.3. Códigos ISO

La ISO (International Organization for Standardization) y la IEC (International Electrotechnical Commission) han desarrollado unas normas llamadas «ISO 8859-X», donde «X» es un número comprendido entre 1 y 16 para adaptar el juego de caracteres extendido a diferentes necesidades:

- ISO 8859-1 (o «Latin-1»), para Europa occidental
- ISO 8859-2 (o «Latin-2»), para Europa central
- ISO 8859-3 (o «Latin-3»), para Europa del sur (Turquía, Malta y Esperanto)
- ...

- ISO 8859-15 (o «Latin-9»), revisión de 8859-1: sustituye ocho símbolos poco usados por € y por algunos caracteres de francés, finés y estonio: Ž, Š, š, ž, Œ, œ, Ÿ
- ISO 8859-16 (o «Latin-10»), para Europa del sudeste (Albania, Croacia, etc.)

Todas ellas comparten las 128 primeras codificaciones con ASCII. La tabla 2.3 reproduce ISO Latin-9. Las treinta y dos primeras codificaciones por encima de ASCII (0x80 a 0x8F) son también caracteres de control dependientes de la aplicación, y no se definen en la norma. «NBSP» (0xA0) significa «non-breaking space» (espacio inseparable) y su interpretación depende de la aplicación. En HTML (un lenguaje que estudiaremos en el Tema 3), por ejemplo, se utilizan varios «` `» (o «` `», o «` `») seguidos cuando se quiere que el navegador presente todos los espacios sin colapsarlos en uno solo.

Hex.	Dec.		Hex.	Dec.		Hex.	Dec.		Hex.	Dec.		Hex.	Dec.		Hex.	Dec.	
A0	160	NBSP	B0	176	°	C0	192	À	D0	208	Đ	E0	224	à	F0	240	ð
A1	161	¡	B1	177	±	C1	193	Á	D1	209	Ñ	E1	225	á	F1	241	ñ
A2	162	¢	B2	178	²	C2	194	Â	D2	210	Ò	E2	226	â	F2	242	ò
A3	163	£	B3	179	³	C3	195	Ã	D3	211	Ó	E3	227	ã	F3	243	ó
A4	164	€	B4	180	Ž	C4	196	Ä	D4	212	Ô	E4	228	ä	F4	244	ô
A5	165	¥	B5	181	μ	C5	197	Å	D5	213	Õ	E5	229	å	F5	245	õ
A6	166	Š	B6	182	¶	C6	198	Æ	D6	214	Ö	E6	230	æ	F6	246	ö
A7	167	§	B7	183	·	C7	199	Ç	D7	215	×	E7	231	ç	F7	247	÷
A8	168	š	B8	184	ž	C8	200	È	D8	216	Ø	E8	232	è	F8	248	ø
A9	169	©	B9	185	¹	C9	201	É	D9	217	Ù	E9	233	é	F9	249	ù
AA	170	ª	BA	186	º	CA	202	Ê	DA	218	Ú	EA	234	ê	FA	250	ú
AB	171	«	BB	187	»	CB	203	Ë	DB	219	Û	EB	235	ë	FB	251	û
AC	172	¬	BC	188	Œ	CC	204	Ì	DC	220	Ü	EC	236	ì	FC	252	ü
AD	173	-	BD	189	œ	CD	205	Í	DD	221	Ý	ED	237	í	FD	253	ý
AE	174	®	BE	190	Ÿ	CE	206	Î	DE	222	Þ	EE	238	î	FE	254	þ
AF	175	¯	BF	191	¿	CF	207	Ï	DF	223	ß	EF	239	ï	FF	255	ÿ

Tabla 2.3 Códigos ISO 8859-15 que extienden ASCII.

Estos estándares son aún muy utilizados, pero paulatinamente están siendo sustituidos por Unicode. La ISO disolvió en 2004 los comités que se encargaban de ellos para centrarse en el desarrollo de Unicode, adoptado con el nombre oficial «ISO/IEC 10646».

2.4. Unicode

El Consorcio Unicode, en colaboración con la ISO, la IEC y otras organizaciones, desarrolla desde 1991 el código UCS (Universal Character Set), que actualmente contiene más de cien mil caracteres. Cada carácter está identificado por un nombre y un número entero llamado **punto de código**.

La primera versión (Unicode 1.1, 1991) definió el «BMP» (Basic Multilingual Plane). Es un código de 16 bits que permite representar $2^{16} = 65.536$ caracteres, aunque no todos los puntos de código se asignan a caracteres, para facilitar la conversión de otros códigos y para expansiones futuras. Posteriormente se fueron añadiendo «planos» para acomodar lenguajes exóticos, y la última versión

(Unicode 6.1, 2012) tiene 17 planos, lo que da un total de $17 \times 2^{16} = 1.114.112$ puntos de código. De ellos, hay definidos 110.118 caracteres. Pero, salvo en países orientales, prácticamente solo se utiliza el BMP.

Ahora bien, una cosa son los puntos de código y otra las codificaciones en binario de esos puntos. El estándar Unicode define siete formas de codificación. Algunas tienen longitud fija. Por ejemplo, «UCS-2» es la más sencilla: directamente codifica en dos bytes cada punto de código del BMP. Pero la más utilizada es «UTF-8», que es compatible con ASCII y tiene varias ventajas para el procesamiento de textos («UTF» es por «Unicode Transformation Format»; otras formas son UTF-16 y UTF-32).

UTF-8

La tabla 2.4 resume el esquema de codificación UTF-8 para el BMP. «U+» es el prefijo que se usa en Unicode para indicar «hexadecimal»; es equivalente a «0x».

Puntos	Punto en binario	Bytes UTF-8
U+0000 a U+007F	00000000 0xxxxxxx	0xxxxxxx
U+0080 a U+07FF	00000yyy yyxxxxxx	110yyyyy 10xxxxxx
U+0800 a U+FFFF	zzzzyyyy yyxxxxxx	1110zzzz 10yyyyyy 10xxxxxx

Tabla 2.4 Codificación en UTF-8.

Los 128 primeros puntos de código se codifican en un byte, coincidiendo con las codificaciones ASCII. Los siguientes 1.920, entre los que están la mayoría de los caracteres codificados en las normas ISO, necesitan dos bytes. El resto de puntos del BMP se codifican en tres bytes. Los demás planos (puntos U+10000 a U+10FFFF, no mostrados en la tabla) requieren 4, 5 o 6 bytes.

	Nombre Unicode	Punto de código	UTF-8
E	LATIN CAPITAL LETTER E	U+0045	0x45
ñ	LATIN SMALL LETTER N WITH TILDE	U+00F1	0xC3B1
₧	PESETA SIGN	U+20A7	0xE282A7
€	EUROSIGN	U+20AC	0xE282AC
⇒	RIGHTWARDS DOUBLE ARROW	U+21D2	0xE28792
∞	INFINITY	U+2210	0xE2889E
∫	INTEGRAL	U+222B	0xE288AB

Tabla 2.5 Ejemplos de codificación en UTF-8.

Algunos ejemplos se muestran en la tabla 2.5. Observe que:

- «E» está codificado en un solo byte, como en ASCII.
- «ñ» no está en ASCII; el byte menos significativo de su punto de código coincide con la codificación en ISO Latin-9 (tabla 2.3), y, como puede usted comprobar, su codificación en UTF-8 se obtiene siguiendo la regla de la tabla 2.4.

- Sin embargo, «€», que también está en la tabla 2.3, tiene un punto de código superior a U+07FF y se codifica en tres bytes. El motivo es que los puntos U+00A0 a U+00FF son compatibles con ISO Latin-1, no con ISO Latin-9, y en el primero la codificación 0xA4 no está asignada al símbolo del euro, sino al símbolo monetario genérico, «¤».

2.5. Cadenas

En Informática se llama «cadena» *(string)* a una secuencia finita de símbolos tomados de un conjunto finito llamado alfabeto.

La representación de un texto plano, una vez elegido un código para la representación de los caracteres es, simplemente, la cadena de bytes que corresponden a la sucesión de caracteres del texto. A diferencia de otros tipos de datos, las cadenas tienen longitud variable. Un convenio que siguen algunos sistemas y lenguajes es indicar al principio de la cadena su longitud (un entero representado en uno o varios bytes). Pero el convenio más frecuente es utilizar una codificación de ASCII, 0x00 («NUL»), para indicar el fin de la cadena. Se les llama «cadenas C» *(C strings)* por ser el convenio del lenguaje C.

Como se trata de una sucesión de bytes, en principio no se plantea el dilema del «extremismo mayor o menor» (apartado 1.4): si el primer byte se almacena en la dirección d, el siguiente lo hará en $d + 1$ y así sucesivamente. Sin embargo:

- Esto es cierto para ASCII y para las normas ISO. Pero en UTF-16 el elemento básico de representación ocupa dos bytes (y en UTF-32, cuatro), y UTF-8 es de longitud variable, y el problema sí aparece. El consorcio Unicode ha propuesto un método que consiste en añadir al principio de la cadena una marca de dos bytes llamada «BOM» (Byte Order Mark) cuya lectura permite identificar si es extremista mayor o menor.

- Se suele ilustrar el conflicto del extremismo con el «problema nUxi», que tiene un origen histórico: al transportar uno de los primeros Unix de un ordenador (PDP-11) a otro (IBM Series/1), un programa que debía escribir «Unix» escribía «nUxi». La explicación es que para generar los cuatro caracteres se utilizaban dos palabras de 16 bits, y el problema surge al almacenar estas palabras en memoria: el programa seguía el convenio del PDP-11 (extremista menor), mientras que el ordenador que lo ejecutaba era extremista mayor.

Capítulo 3

Representación de datos numéricos

Los datos de tipo numérico pueden codificarse en binario siguiendo distintos métodos. Los más usuales están basados en uno de dos principios: o bien codificar cada dígito decimal mediante un código BCD (de modo que un número de n dígitos decimales se codifique con n cuartetos, apartado 2.1), o bien representar el número en el sistema de numeración de base 2. En cualquier caso, hay que añadir convenios para representar números negativos y números no enteros.

La codificación en BCD actualmente solo se usa en dispositivos especializados (relojes, calculadoras, etc.). En la asignatura «Electrónica digital» estudiará usted algunos casos.

Por otra parte, se pueden representar números con *precisión arbitraria*, utilizando cuantos bits sean necesarios (con la limitación de la capacidad de memoria o del caudal del canal de transmisión disponibles, por eso no es «precisión infinita»). Hay lenguajes y bibliotecas de programas que permiten representar de este modo y realizar operaciones aritméticas *(«bignum arithmetic»)* para aplicaciones especiales, pero tampoco nos ocuparemos de este tipo de representación.

Nos vamos a centrar en los convenios normalmente adoptados en los diseños de los procesadores de uso general. Se basan en reservar un número fijo de bits (normalmente, una palabra, o un número reducido de palabras) para representar números enteros y racionales con una precisión y un rango (números máximo y mínimo) determinados por el número de bits. La figura 3.1 generaliza a n bits otra que ya habíamos visto (figura 1.1).

El bit 0 (o «bit de peso cero») es el *bit menos significativo, «bms»* (o «LSB»: Least Significant Bit) y el de peso $n-1$ es el *bit más significativo, «bMs»* (o «MSB»: Most Significant Bit).

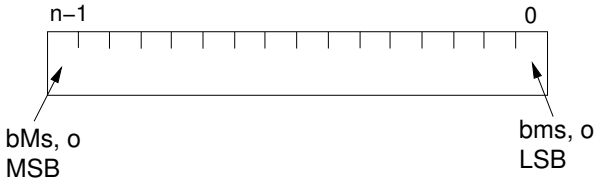

Figura 3.1 «Contenedor» de n bits.

3.1. Números enteros sin signo

Algunos datos son de tipo entero no negativo, como las direcciones de la memoria, o el número de segundos transcurridos a partir de un instante dado. Su representación es, simplemente, su expresión en binario. Con n bits el máximo número representable es $0b111\ldots1 = 2^n - 1$ («0b» indica que lo que sigue está en el sistema de numeración de base 2). El *rango* de la representación es $[0, 2^n - 1]$.

Por ejemplo, para expresar las direcciones de una memoria de 1 MiB = 2^{20} bytes necesitamos $n = 20$ bits. La primera dirección será 0x00000 y la más alta 0xFFFFF = $2^{20} - 1$. Otro ejemplo: las direcciones IPv4 son enteros sin signo de 32 bits. La más pequeña sería 0x00000000 (0.0.0.0) y la más grande 0xFFFFFFFF (255.255.255.255). En total, 2^{32} = 4.294.967.296 direcciones.

3.2. Formatos de coma fija

En el contexto de la representación de números, un **formato** es la definición del significado de cada uno de los n bits dedicados a la representación. Se distinguen en él partes o **campos**, grupos de bits con significado propio. Los formatos más utilizados para la representación de datos numéricos son los de **coma fija** y **coma flotante**.

Los formatos de coma fija solo tienen dos campos. Uno es el bit más significativo o **bit de signo**, bMs = S: si S = 0, el número representado es positivo, y si S = 1, es negativo. El campo formado por los $n - 1$ bits restantes representa la magnitud del número. Se siguen distintos convenios sobre cómo se codifica esta magnitud.

3.3. Números enteros

Al representar un número entero en un formato de coma fija, la coma se supone situada inmediatamente a la derecha del bms.

Como el primer bit es el de signo, el máximo número representable (positivo) es $0b0111\ldots11 = 2^{n-1} - 1$. El mínimo depende del convenio para los números negativos.

Convenios para números negativos

Hay tres convenios:

- **signo y magnitud** (o **signo y módulo**): después del signo (S = 1), el valor absoluto;
- **complemento a 1**: se cambian los «ceros» por «unos» y los «unos» por «ceros» en la representación binaria del correspondiente número positivo;
- **complemento a 2**: se suma una unidad al complemento a 1.

Por ejemplo, el número –58 = –0x3A = –0b111010, con $n = 16$ bits, y según el convenio que se adopte, da lugar a las siguientes representaciones:

- con signo y módulo: 1000000000111010 (0x803A)
- con complemento a 1: 1111111111000101 (0xFFC5)
- con complemento a 2: 1111111111000110 (0xFFC6)

Es fácil comprobar que si X es la representación de un número con n bits, incluido el de signo, y si $X1$ y $X2$ son las representaciones en complemento a 1 y complemento a 2, respectivamente, del mismo número cambiado de signo, entonces $X + X1 = 2^n - 1$, y $X + X2 = 2^n$. (Por tanto, en rigor, lo que habitualmente llamamos «complemento a 2» es «complemento a 2^n», y «complemento a 1» es «complemento a $2^n - 1$»).

La tabla 3.1 muestra una comparación de los tres convenios para n bits. Como puede usted observar, en signo y magnitud y en complemento a 1 existen dos representaciones para el número 0, mientras que en complemento a 2 la representación es única (y hay un número negativo más, cuyo valor absoluto no puede representarse con n bits). El *rango*, o *extensión* de la representación (diferencia entre el número máximo y el mínimo representables) es una función exponencial de n.

En lo sucesivo, y salvo en alguna nota marginal, *supondremos siempre que el convenio es el de complemento a 2*, que es el más común.

	configuraciones binarias	interpretaciones decimales		
		signo y magnitud	complemento a 1	complemento a 2
números positivos	0000…0000	0	0	0
	0000…0001	1	1	1
	…	…	…	…
	0111…1110	$2^{n-1}-2$	$2^{n-1}-2$	$2^{n-1}-2$
	0111…1111	$2^{n-1}-1$	$2^{n-1}-1$	$2^{n-1}-1$
números negativos	1000…0000	0	$-(2^{n-1}-1)$	$-(2^{n-1})$
	1000…0001	-1	$-(2^{n-1}-2)$	$-(2^{n-1}-1)$
	…	…	…	…
	1111…1110	$-(2^{n-1}-2)$	-1	-2
	1111…1111	$-(2^{n-1}-1)$	0	-1

Tabla 3.1 Comparación de convenios para formatos de coma fija.

3.4. Operaciones básicas de procesamiento

En este Tema estamos estudiando los convenios de representación de varios tipos de datos. Es posible que, a veces, alguno de esos convenios le parezca a usted arbitrario. Por ejemplo, ¿por qué representar en complemento los números negativos, cuando parece más «natural» el hacerlo con signo y módulo?

Pero tenga en cuenta que los convenios no se adoptan de forma caprichosa, sino guiada por el tipo de operaciones que se realizan sobre los datos y por los algoritmos o por los circuitos electrónicos que implementan estas operaciones en software o en hardware. Aunque nos desviemos algo del contenido estricto del tema («representación»), éste es un buen momento para detenernos en algunas operaciones básicas de *procesamiento*.

Para no perdernos con largas cadenas de bits, vamos a basarnos en ejemplos muy simplificados, en el sentido de un número de bits muy reducido: $n = 6$. Esto daría un rango de representación para números enteros sin signo de $[0, 63]$ y un rango para enteros de $[-32, 31]$, lo que sería insuficiente para la mayoría de las aplicaciones prácticas, pero es más que suficente para ilustrar las ideas.

Suma y resta de enteros sin signo

La suma binaria se realiza igual que la decimal: primero se suman los bits de peso 0, luego los de peso 1 más el eventual acarreo (que se habrá producido en el caso de 1+1=10), etc. Por ejemplo:

```
  13    001101    (0x0D)
+39   +100111   (+0x27)
  52    110100    (0x34)
```

Observe que se puede producir acarreo en todos los bits salvo en el más significativo, porque eso indica que el resultado no puede representarse en el rango, y se dice que hay un **desbordamiento**. Por ejemplo:

```
  60     111100   (0x3C)
+15    +001111   (+0x0F)
  75  1) 001011   (0x4B, pero el máximo es 0x3F: desbordamiento)
```

Para la resta también se podría aplicar un algoritmo similar al que conocemos para la base 10, pero para el diseño de circuitos es más fácil trabajar con complementos. Como estamos tratando de

enteros *sin signo* con seis bits, no tiene sentido el complemento a 2^6, pero si sumamos al minuendo el complemento a 2^7 del sustraendo obtenemos el resultado correcto. Por ejemplo:

$$\begin{array}{rrl} 60 & 111100 & (0x3C) \\ \underline{-15} & \underline{+1)110001} & (\underline{+0x71}) \text{ (complemento a } 2^7 \text{de 15)} \\ 45 & 10)\ 101101 & (0x2D, \text{ olvidando el acarreo)} \end{array}$$

Fíjese que ahora *siempre hay un acarreo* de los bits más significativos que hay que despreciar, incluso si el resultado es 0 (puede comprobarlo fácilmente).

Parece que no tiene mucho sentido preguntarse por el caso de que el sustraendo sea mayor que el minuendo, ya estamos tratando de números positivos, pero a veces lo que interesa es *comparar* dos números: saber si X es mayor, igual o menor que Y, y esto se puede averigurar sumando a X el complemento de Y. Por ejemplo:

$$\begin{array}{rrl} 15 & 001111 & (0x0F) \\ \underline{-60} & \underline{+1)000100} & (+0x44) \text{ (complemento a } 2^7 \text{de 60)} \\ & 010011 & \end{array}$$

Lo importante ahora no es el resultado, obviamante incorrecto, sino el hecho de que *no hay acarreo de los bits más significativos*.

Conclusión: para comparar X e Y, representados sin signo en n bits, sumar a X el complemento a 2^{n+1} de Y. Si se produce acarreo de los bits más significativos, $X \geq Y$; en caso contrario, $X < Y$.

Suma y resta de enteros

La representación de los números negativos por su complemento conduce a algoritmos más fáciles para la suma y resta que en el caso de signo y módulo (y, consecuentemente, a circuitos electrónicos más sencillos para su implementación).

Como hemos visto un poco antes, si X es un entero positivo y $X2$ es su complemento a 2, $X + X2 = 2^n$.

Por ejemplo:
$$\begin{array}{rrl} 7 & 000111 & \\ \underline{+(-7)} & \underline{+111001} & \\ 0 & 1000000 & (2^6) \end{array}$$

Veamos cómo se aplica esta propiedad a la resta. Sean X y Y dos números positivos. La diferencia $D = X - Y$ se puede escribir así:
$D = X - Y = X - (2^n - Y2) = X + Y2 - 2^n$
donde $Y2$ es el complemento a 2 de Y.

Analicemos dos posibilidades: o bien $X + Y2 \geq 2^n$ ($D \geq 0$), o bien $X + Y2 < 2^n$ ($D < 0$).

- Si $D \geq 0$, restar 2^n de $X + Y2$ es lo mismo que quitar el bit de peso n (acarreo de los bits de signo).
- Si $D < 0$ la representación de D en complemento a 2 será $D2 = 2^n - (-D) = X + Y2$.

O sea, la suma binaria $X+Y2$, olvidando el posible acarreo de los bits de signo, nos va a dar siempre el resultado correcto de $X - Y$.

¿Y si el minuendo es negativo? La operación $L = -X - Y$ (siendo X y Y positivos) se escribe así en función de los complementos:

$L = -X - Y = -(2^n - X2) - (2^n - Y2) = X2 + Y2 - 2^{n+1}$

El resultado ha de ser siempre negativo (a menos, como veremos luego, que haya desbordamiento), es decir, $X2 + Y2 < 2^{n+1}$. La representación en complemento a 2 de este resultado es:

$L2 = 2^n - (-L) = X2 + Y2 - 2^n$.

En conclusión, *para restar dos números basta sumar al minuendo el complemento a 2 del sustraendo y no tener en cuenta el eventual acarreo de los bits de signo*[1]. Ejemplos:

- $\begin{array}{rrl} 25 & 011001 & \\ \underline{-7} & \underline{+111001} & \\ 18 & 1010010 & \rightsquigarrow \quad 010010 \text{ (representación de 18)} \end{array}$

- $\begin{array}{rrl} 14 & 001110 & \\ \underline{-30} & \underline{+100010} & \\ -16 & 110000 & \quad (-16 \text{ en complemento a 2}) \end{array}$

- $\begin{array}{rrl} -10 & 110110 & \\ \underline{-18} & \underline{+101110} & \\ -28 & 1100100 & \rightsquigarrow \quad 100100 \ (-28 \text{ en complemento a 2}) \end{array}$

Desbordamiento

En todos los ejemplos anteriores (salvo en uno) hemos supuesto que tanto los operandos como el resultado estaban dentro del rango de números representables, que en el formato de los ejemplos es $-32 \leq X \leq 31$. Si no es así, las representaciones resultantes son erróneas. Por ejemplo:

- $\begin{array}{rrl} 15 & 001111 & \\ \underline{+20} & \underline{+010100} & \\ 35 & 100011 & \quad (-29 \text{ en complemento a 2}) \end{array}$

- $\begin{array}{rrl} -13 & 110011 & \\ \underline{-27} & \underline{+100101} & \\ -40 & 1011000 & \rightsquigarrow \quad 011000 \ (+24) \end{array}$

Este **desbordamiento** solo se produce cuando los dos operandos son del mismo signo (se supone, por supuesto, que ambos están dentro del rango representable). Y su detección es muy fácil: cuando lo hay, el resultado es de signo distinto al de los operandos[2].

Acarreo

A veces el bit de signo no es un bit de signo, ni el desbordamiento indica desbordamiento. Veamos cuáles son esas «veces».

Supongamos que tenemos que sumar 1075 (0x433) y 933 (0x3A5). El resultado debe ser 2008 (0x7D8). Para hacerlo en binario necesitamos una extensión de al menos doce bits:

$$\begin{array}{rcl} 0\text{x}433 & \rightsquigarrow & 010000110011 \\ 0\text{x}3\text{A}5 & \rightsquigarrow & \underline{001110100101} \\ & & 011111011000 \quad \rightsquigarrow \quad 0\text{x}7\text{D}8 \end{array}$$

Pero supongamos también que nuestro procesador solamente permite sumar con una extensión de seis bits. Podemos resolver el problema en tres pasos:

[1] Si se entretiene usted en hacer un análisis similar para el caso de complemento a 1, descubrirá que se hace necesario un paso más: el eventual acarreo de los bits de signo no se debe despreciar, sino sumar a los bits de peso 0 para que el resultado sea correcto.

[2] Una condición equivalente (y más fácil de comprobar en los circuitos, aunque su enunciado sea más largo) es que el acarreo de la suma de los bits de pesos inmediatamente inferior al de signo es diferente al acarreo de los bits de signo (el que se descarta).

1. Sumar los seis bits menos significativos como si fuesen números sin signo:
   $$\begin{array}{r} 110011 \\ +100101 \\ \hline 1\,011000 \end{array}$$ (los seis bits menos significativos del resultado)
   Obtenemos un resultado en seis bits y un acarreo de los bits más significativos. Observe que esta suma es justamente la que provocaba desbordamiento en el ejemplo anterior de $-13-27$, pero ahora no estamos interpretando esos seis bits como la representación de un número entero con signo: los bits más significativos no son «bits de signo».
2. Sumar los seis bits más significativos:
   $$\begin{array}{r} 010000 \\ +001110 \\ \hline 011110 \end{array}$$
3. Al último resultado, sumarle el acarreo de los seis menos significativos:
   $$\begin{array}{r} 011110 \\ +1 \\ \hline 011111 \end{array}$$ (los seis bits más significativos del resultado)

El ejemplo ha tratado de la suma de dos números positivos, pero el principio funciona exactamente igual con números negativos representados en complemento.

Veremos en el siguiente Tema que los procesadores suelen tener dos instrucciones de suma: suma «normal» (que suma operandos de, por ejemplo, 32 bits) y suma con acarreo, que suma también dos operandos, pero añadiendo el eventual acarreo de la suma anterior.

Indicadores

Los circuitos de los procesadores (unidades aritmética y lógica y de desplazamiento) incluyen indicadores de un bit (materializados como biestables, apartado 1.3) que toman el valor 0 o 1 dependiendo del resultado de la operación. Normalmente son cuatro:

C: acarreo. Toma el valor 1 si, como acabamos de ver, se produce acarreo en la suma o resta sin signo. Y como vimos antes, indica también, en la resta de dos números sin signo, $X - Y$, si $X \geq Y$.

V: desbordamiento. Toma el valor 1 si se produce un desbordamiento en la suma o en la resta con signo.

N: negativo. Coincide con el bit más significativo del resultado (S). Si no se ha producido desbordamiento, N = 1 indica que el resultado es negativo.

Z: cero. Toma el valor 1 si todos los bits del resultado son 0.

N y V, conjuntamente, permiten detectar el mayor de dos enteros *con signo* al hacer la resta $X - Y$:

- $X \geq Y$ si N = V, o sea, si N = 0 y V = 0 o si N = 1 y V = 1
- $X < Y$ si N $\neq$ V, o sea, si N = 0 y V = 1 o si N = 1 y V = 0

Operaciones lógicas

Las principales operaciones lógicas son la *complementación* o *negación* (**NOT**), el *producto lógico* (**AND**), la *suma lógica* (**OR**), la *suma módulo 2* u *or excluyente* (**EOR**), el *producto negado* (**NAND**) y la *suma negada* (**NOR**). Se trata de las mismas operaciones del álgebra de Boole, aplicadas en paralelo a un conjunto de bits. Aunque usted ya las debe conocer por la asignatura de Álgebra, las resumimos brevemente.

La primera se aplica a un solo operando y consiste, simplemente, en cambiar todos sus «ceros» por «unos» y viceversa. Si el operando se interpreta como la representación de un número entero, el resultado es su complemento a 1. Por ejemplo: NOT(01101011) = 10010100

Las otras cinco se aplican, bit a bit, sobre dos operandos, y se definen como indica la tabla 3.2, donde en las dos primeras columnas se encuentran los cuatro valores posibles de dos operandos de un bit, y en las restantes se indican los resultados correspondientes a cada operación.

op1	op2	AND	OR	EOR	NAND	NOR
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	1	0
1	1	1	1	0	0	0

Tabla 3.2 Operaciones lógicas sobre dos operandos.

El **enmascaramiento** consiste en realizar la operación *AND* entre un dato binario y un conjunto de bits prefijado (**máscara**) para poner a cero determinados bits (y dejar los demás con el valor que tengan). Así, para extraer los cuatro bits menos significativos de un byte utilizaremos la máscara 0x0F = 00001111, y si la información original es «abcdefgh» (a, b,... ,h ∈ {0,1}) el resultado será «0000efgh».

Operaciones de desplazamiento

Los desplazamientos, lo mismo que las operaciones lógicas, se realizan sobre un conjunto de bits, normalmente, una palabra. Hay dos tipos básicos: *desplazamientos a la derecha*, en los que el bit *i* del operando pasa a ocupar la posición del bit $i - 1$, éste la del $i - 2$, etc., y *desplazamientos a la izquierda*, en los que se procede al revés. Y hay varios subtipos (figura 3.2), dependiendo de lo que se haga con el bit que está más a la derecha (bms) y con el que está más a la izquierda (bMs):

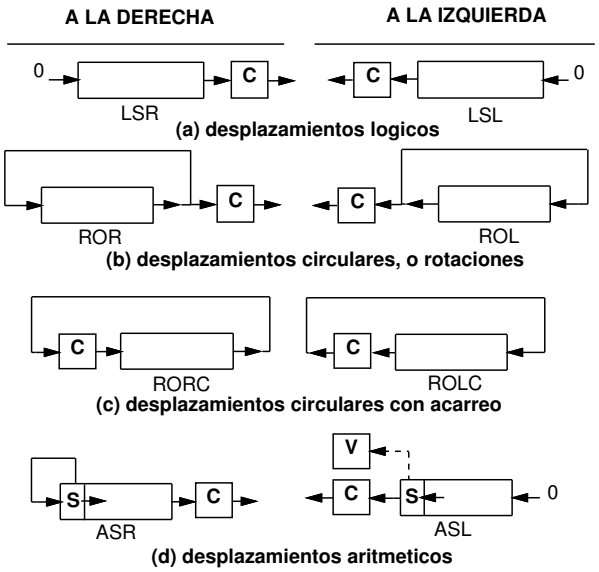

Figura 3.2 Operaciones de desplazamiento.

Desplazamientos lógicos

En un desplazamiento lógico a la derecha el bms se introduce en el indicador C (perdiéndose el valor anterior que éste tuviese), y por la izquierda se introduce un «0» en el bMs. Justamente lo contrario se hace en el desplazamiento lógico a la izquierda: el bMs se lleva a C y se introduce un «0» por la derecha, como ilustra la figura 3.2(a).

Las siglas que aparecen en la figura son nombres nemónicos en inglés:
`LSR` es «Logical Shift Right» y `LSL` es «Logical Shift Left».

Desplazamientos circulares

En los desplazamientos circulares, también llamados **rotaciones**, el bit que sale por un extremo se introduce por el otro, como indica la figura 3.2(b). Los nemónicos son `ROR` («Rotate Right») y `ROL` («Rotate Left»).

Las rotaciones a través del indicador C funcionan como muestra la figura 3.2(c).

Desplazamientos aritméticos

En el desplazamiento aritmético a la izquierda, el bMs se introduce en el indicador de acarreo (C), y en el desplazamiento aritmético a la derecha se propaga el bit de signo. La figura 3.2(d) ilustra este tipo de desplazamiento, muy útil para los algoritmos de multiplicación. Es fácil comprobar que un desplazamiento aritmético de un bit a la derecha de un entero con signo conduce a la representación de la división (entera) por 2 de ese número. Y que el desplazamiento a la izquierda hace una multiplicación por 2.

Los nemónicos son `ASR` («Arithmetic Shift Right») y `ASL` («Arithmetic Shift Left»).

Aparentemente, el desplazamiento aritmético a la izquierda tiene el mismo efecto que el desplazamiento lógico a la izquierda, pero hay una diferencia sutil: el primero pone un «1» en el indicador de desbordamiento (V) si como consecuencia del desplazamiento cambia el bit de signo (S).

3.5. Números racionales

Volvamos al asunto principal de este Tema, que es la representación de datos.

Para la representación de números racionales con una cantidad finita de bits hemos de tener en cuenta que no solo habrá una **extensión** finita (un número máximo y un número mínimo), sino también una **resolución** (diferencia entre dos representaciones consecutivas) finita, es decir, hay una *discretización* del espacio continuo de los números reales.

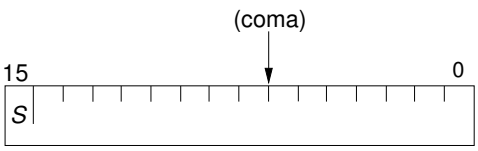

Figura 3.3. Formato de 16 bits para representación en coma fija de números racionales.

En principio, con un formato de coma fija podemos representar también números racionales: basta con añadir el convenio sobre «dónde está la coma». Si, por ejemplo, tenemos una longitud de dieciséis bits, podríamos convenir que los ocho que siguen al bit de signo representan la parte entera y los siete restantes la parte fraccionaria (figura 3.3).

En este ejemplo, el máximo número representable es 0b0 1111 1111,1111 111 = +0xFF,FE = +255,9921875 y el mínimo, suponiendo complemento a 2, es el representado por 1000000000000000, que corresponde a –0b1 0000 0000,0000 000 = –0x100 = –256. La resolución es: 0b0,0000 001 = 0x0,02 = 0,0078125.

En general, si tenemos f bits para la parte fraccionaria, estamos aplicando un factor de escala $1/2^f$ con respecto al número entero que resultaría de considerar la coma a la derecha. El máximo

representable es $(2^{n-1} - 1)/2^f$, y el mínimo, $-2^{n-1}/2^f$. La diferencia máx − mín $= 2^{-f}$ es igual a la resolución.

Pero la rigidez que resulta es evidente: para conseguir la máxima extensión, en cada caso tendríamos que decidir dónde se encuentra la coma, en función de que estemos trabajando con números muy grandes o muy pequeños. Aun así, la extensión que se consigue puede ser insuficiente: con 32 bits, el número máximo representable (todos «unos» y la coma a la derecha del todo) sería $2^{31} - 1 \approx 2 \times 10^9$, y el mínimo positivo (coma a la izquierda y todos «ceros» salvo el bms) sería $2^{-31} \approx 2 \times 10^{-10}$

Por esto, lo más común es utilizar un formato de coma flotante, mucho más flexible y con mayor extensión.

Formatos de coma flotante

El principio de la representación en coma flotante de un número X consiste en codificar dos cantidades, A y E, de modo que el número representado sea:

$$X = \pm A \times b^E$$

Normalmente, $b = 2$. Solo en algunos ordenadores se ha seguido el convenio de que sea $b = 16$.

En un formato de coma flotante hay tres campos (figura 3.4):

- S: *signo* del número, 1 bit.
- M: *mantisa*, m bits.
- C: *característica*, e bits.

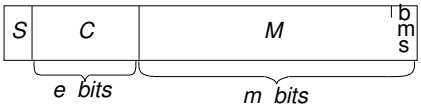

Figura 3.4. Formato de coma flotante.

Se pueden seguir varios convenios para determinar cómo se obtienen los valores de A y E en función de los contenidos binarios de M y C

Mantisa

Los m bits de la mantisa representan un número en coma fija, A. Hace falta un convenio sobre la parte entera y la fraccionaria (o sea «dónde está la coma»). Normalmente la representación está «normalizada». Esto quiere decir que el exponente se ajusta de modo que la coma quede a la derecha del bit menos significativo de M (normalización entera) o a la izquierda del más significativo (normalización fraccionaria). Es decir:

- Normalización entera: $X = \pm M \times 2^E$
- Normalización fraccionaria: $X = \pm 0, M \times 2^E$, o bien:
  $X = \pm 1, M \times 2^E$

Normalizar es trivial: por cada bit que se desplaza la coma a la izquierda o a la derecha se le suma o se le resta una unidad, respectivamente, a E.

Veamos con un ejemplo por qué conviene que la representación esté normalizada. Para no perdernos con cadenas de bits, supongamos por esta vez, para este ejemplo, que la base es $b = 16$. Supongamos también que el formato reserva $m = 24$ bits para la mantisa (24/4 = 6 dígitos hexadecimales), y que se trata de representar el número π :

$X = \pi = 3{,}1415926\ldots = 0\text{x}3{,}243F69A\ldots$

Consideremos primero el caso de normalización entera; tendríamos las siguientes posibilidades:

M	E	Número representado
0x000003	0	$0x3\times16^{0} = 0x3 = 3$
0x000032	-1	$0x32\times16^{-1} = 0x3{,}2 = 3{,}125$
0x000324	-2	$0x324\times16^{-2} = 0x3{,}24 = 3{,}140625$
0x003243	-3	$0x3243\times16^{-3} = 0x3{,}243 = 3{,}1413574$
0x03243F	-4	$0x3243F\times16^{-4} = 0x3{,}243F = 3{,}1415863$
0x3243F6	-5	$0x3243F6\times16^{-5} = 0x3{,}243F6 = 3{,}141592$

La última es la representación normalizada; es la que tiene más dígitos significativos, y, por tanto, mayor precisión.

Con normalización fraccionaria y el convenio $X = \pm 0, M \times 16^{E}$:

M	E	Número representado
0x000003	6	$0x0{,}000003\times16^{6} = 0x3 = 3$
0x000032	5	$0x0{,}000032\times16^{5} = 0x3{,}2 = 3{,}125$
...	...	...
0x3243F6	1	$0x0{,}3243F6\times16^{1} = 0x3{,}243F6 = 3{,}141592$

y, como antes, la última es la representación normalizada.

El número «0» no se puede normalizar. Se conviene en representarlo por todos los bits del formato nulos.

Para los números negativos se pueden seguir también los convenios de signo y magnitud, complemento a 1 o complemento a 2 de la mantisa.

Exponente

En los e bits reservados para el exponente podría seguirse también el convenio de reservar uno para el signo y los $e-1$ restantes para el módulo, o el complemento a 1 o a 2 si $E < 0$. Pero el convenio más utilizado no es éste, sino el de la representación *en exceso de* 2^{e-1}: el número *sin signo* contenido en esos e bits, llamado **característica**, es:

$$C = E + 2^{e-1}$$

Dado que $0 \leq C \leq 2^{e} - 1$, los valores representables de E estarán comprendidos entre -2^{e-1} (para $C = 0$) y $2^{e-1} - 1$ (para $C = 2^{e} - 1$).

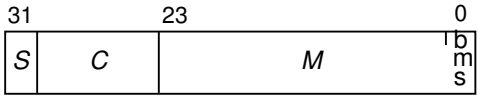

Figura 3.5. Un formato de coma flotante con 32 bits.

Como ejemplo, veamos la representación del número π en el formato de la figura 3.5, suponiendo que el exponente está en exceso de 64 con base 16 y que la mantisa tiene normalización fraccionaria. De acuerdo con lo que hemos explicado antes, la mantisa será:
0x3243F6 = 0b0011001001000011111110110.
Para la característica tendremos: $C = 1 + 64 = 65 = 0x41 = 0b1000001$, y la representación resulta ser:
0 1000001 0011 0010 0100 0011 1111 0110

Observe que, al ser la base 16, la mantisa está normalizada en hexadecimal, pero no en binario, ni lo puede estar en este caso, porque cada incremento o decremento de E corresponde a un desplazamiento de cuatro bits en M.

Si cambiamos uno de los elementos del convenio, la base, y suponemos ahora que es 2, al normalizar la mantisa resulta:
0x3,243F6A... =
0b11,001001000011111101101010... =
$0b0{,}11001001000011111101101010...\times2^{2}$

La característica es ahora $C = 2 + 64$. Truncando a 24 los bits de la mantisa obtenemos:
0 1000010 1100 1001 0000 1111 1101 1010
Vemos que al normalizar en binario ganamos dos dígitos significativos en la mantisa.

El mismo número pero negativo ($-\pi$), con convenio de complemento a 2, se representaría en el primer caso ($b = 16$) como:
1 1000001 1100 1101 1011 1100 0000 1010
y en el segundo ($b = 2$):
1 1000010 0011 0110 1111 0000 0010 0110

El inconveniente de la base 16 es que la mantisa no siempre se puede normalizar en binario, y esto tiene dos consecuencias: por una parte, como hemos visto, pueden perderse algunos bits (1, 2 o 3) con respecto a la representación con base 2; por otra, hace algo más compleja la realización de operaciones aritméticas. La ventaja es que, con el mismo número de bits reservados para el exponente, la escala de números representables es mayor. Concretamente, para $e = 7$ el factor de escala, con $b = 2$, va de 2^{-64} a 2^{63} (o, aproximadamente, de 5×10^{-20} a 10^{19}), mientras que con $b = 16$ va de 16^{-64} a 16^{63} (aproximadamente, de 10^{-77} a 10^{76}).

Norma IEEE 754

El estándar IEEE 754 ha sido adoptado por la mayoría de los fabricantes de hardware y de software. Define los formatos, las operaciones aritméticas y el tratamiento de las «excepciones» (desbordamiento, división por cero, etc.). Aquí solo nos ocuparemos de los formatos, que son los esquematizados en la figura 3.6.[3]

Los convenios son:

- Números negativos: signo y magnitud.
- Mantisa: normalización fraccionaria.
- Base implícita: 2.
- Exponente: en exceso de $2^{e-1} - 1$

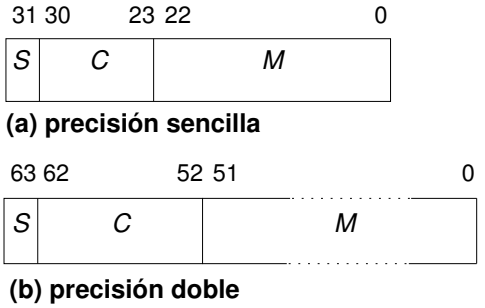

Figura 3.6 Formatos de la norma IEEE 754.

Como la mantisa tiene normalización fraccionaria y los números negativos se representan con signo y magnitud, el primer bit de la mantisa siempre sería 1, lo que permite prescindir de él en la representación y ganar así un bit. La coma se supone inmediatamente a la derecha de ese «1 implícito», por lo que, dada una mantisa M y un exponente E, el número representado es $\pm(1, M) \times 2^E$.

El exponente, E, se representa en exceso de $2^{e-1} - 1$ (no en exceso de 2^{e-1}). La extensión de E en el formato de 32 bits, que tiene $e = 8$, resulta ser: $-127 \leq E \leq 128$, y en el de 64 bits ($e = 11$): $-1.023 \leq E \leq 1.024$. Pero los valores extremos de C (todos ceros o todos unos) se reservan para representar casos especiales, como ahora veremos. Esto nos permite disponer, en definitiva, de un factor de escala que está comprendido entre 2^{-126} y 2^{127} en el formato de 32 bits y entre $2^{-1.022}$ y $2^{1.023}$ en el de 64 bits.

Resumiendo, dados unos valores de M y C, el número representado, X, si se trata del formato de precisión sencilla (figura 3.6(a)), es el siguiente:

[3] La norma contempla otros dos formatos, que son versiones extendidas de éstos: la mantisa ocupa una palabra completa (32 o 64 bits) y la característica tiene 10 o 14 bits. Pero los de la figura 3.6 son los más utilizados.

1. Si $0 < C < 255, X = \pm(1, M) \times 2^{C-127}$
2. Si $C = 0$ y $M = 0, X = 0$
3. Si $C = 0$ y $M \neq 0, X = \pm 0, M \times 2^{-126}$
   (números más pequeños que el mínimo representable en forma normalizada)
4. Si $C = 255$ y $M = 0, X = \pm\infty$
   (el formato prevé una representación específica para «infinito»)
5. Si $C = 255$ y $M \neq 0, X =$ NaN
   («NaN» significa «no es un número»; aquí se representan resultados de operaciones como $0/0, 0\times\infty$, etc.)

Y en el formato de precisión doble (figura 3.6(b)):

1. Si $0 < C < 2.047, X = \pm(1, M) \times 2^{C-1.023}$
2. Si $C = 0$ y $M = 0, X = 0$
3. Si $C = 0$ y $M \neq 0, X = \pm 0, M \times 2^{-1.022}$
4. Si $C = 2.047$ y $M = 0, X = \pm\infty$
5. Si $C = 2.047$ y $M \neq 0, X =$ NaN

Una «calculadora» para pasar de un número a su representación y a la inversa se encuentra en `http://babbage.cs.qc.edu/IEEE-754/`

Capítulo 4

Detección de errores y compresión

A diferencia del «mundo analógico», en el que una pequeña perturbación en una señal normalmente pasa desapercibida, en el digital la sola alteración de un bit entre millones puede cambiar totalmente el mensaje.

En algunas aplicaciones (transferencia de ficheros, transacciones bancarias...) no puede admitirse ningún error. En otras (imágenes, comunicaciones de voz, «streaming» multimedia...) basta con reducirlos a un mínimo aceptable.

Hay muchos algoritmos para detectar si se ha producido un error en la transmisión de un mensaje codificado en binario. En general, se basan en *añadir redundancia* al mensaje. Como ilustra la figura 4.1, al mensaje original se le añaden bits de comprobación, que no aportan información para el receptor, pero sirven para verificar si se ha producido o no algún error.

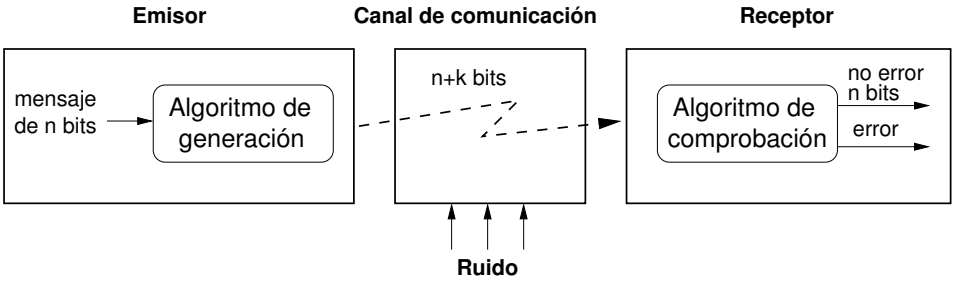

Figura 4.1 Detección de errores.

Hay algoritmos que permiten no solo detectar errores, también pueden corregirlos. Este asunto se estudia en otras asignaturas de Telemática. Veremos aquí el principio de los dos algoritmos de detección más sencillos, y de las acciones posibles cuando se detecta un error.

Después entraremos en un asunto en cierto modo «opuesto» y sin embargo complementario: si la detección de errores se basa en añadir redundancia, la compresión consiste en *reducir redundancia*. Si un mensaje de N bits puede transformarse mediante un algoritmo en otro de N_C bits de modo que $k = N/N_C > 1$ y que no haya pérdida en la información que el agente receptor interpreta (o que la pérdida sea aceptable), se dice que el algoritmo tiene un **factor de compresión** $f_c = k : 1$, o que el **porcentaje de compresión** es $C = 100/k\,\%$.

4.1. Bit de paridad

El algoritmo más básico para detectar errores en un mensaje binario consiste en añadir un bit cada n bits, de modo que el número total de «1» en los $n + 1$ bits sea o bien par («paridad par») o bien impar («paridad impar»).

Como ejemplo, consideremos un mensaje de texto que consiste solamente en la palabra «Hola». Los caracteres se codifican en ASCII (siete bits por carácter) y se transmiten en un orden extremista mayor: primero, «H» (0x48 = 1001000), etc. Si convenimos en añadir un bit de paridad impar, a la codificación de «H» se le añade en el extremo emisor un octavo bit, que en este caso deberá ser «1» para que el número total sea impar. Suponiendo que se conviene en que el bit de paridad se añade en la posición más significativa, para los cuatro caracteres del mensaje resulta esto:

Carácter	ASCII (hex)	ASCII (bin)		Resultado	Resultado (hex.)
H	0x48	1001000	$\rightsquigarrow$	*1*1001000	0xC8
o	0x6F	1101111	$\rightsquigarrow$	*1*1101111	0xEF
l	0x6C	1101100	$\rightsquigarrow$	*1*1101100	0xEC
a	0x61	1100001	$\rightsquigarrow$	*0*1100001	0x61

El receptor, que, naturalmente, debe estar diseñado con los mismos convenios (en este ejemplo, grupos de ocho bits incluyendo el de paridad en la posición más significativa, y paridad impar) comprueba la paridad cada ocho bits. Si en algún grupo hay un número par de «1» señala el error. En caso contrario, quita el bit de paridad, con lo que recupera la codificación original.

La ventaja del procedimiento es su sencillez de implementación: el bit de paridad se calcula en el emisor haciendo una operación lógica «OR excluyente» (apartado 3.2) de todos los bits del grupo, y en el receptor se comprueba del mismo modo. Y esta operación es fácil de realizar con componentes electrónicos, por lo que la incorporan algunos componentes de hardware, como el bus PCI.

Por ejemplo («$\oplus$» significa «or excluyente»):

- E tiene que enviar 0x48 = 1001000
- E calcula el bit de paridad: $1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0$
- Si el convenio es de paridad impar, el bit de paridad es $\overline{0} = 1$
- E envía 11001000
- Hay un error en el bit menos significativo, y R recibe 11001001
- R calcula el bit de paridad: $1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 0$
- Como el convenio es de paridad impar, debería haber resultado 1, por lo que R detecta que ha habido un error.

El inconveniente es que si se alteran dos bits (o, en general, un número par de bits), el algoritmo no lo detecta.

El procedimiento del bit de paridad es un caso degenerado de otros más robustos y muy utilizados en redes (como Ethernet) y en las transferencias con discos: los CRC (Cyclic Redundancy Check). Los CRC tienen una fundamentación matemática: la teoría algebraica de anillos. Si se ve usted obligado estudiar fundamentos matemáticos como éstos u otros piense que no es por martirizarle: en otras asignaturas irá viendo que son imprescindibles para la ingeniería.

4.2. Suma de comprobación *(checksum)*

Los procedimientos basados en sumas de comprobación son algo más robustos y más adecuados para mensajes largos que el del bit de paridad.

En general, una suma de comprobación, o de prueba, o «suma hash», es un conjunto de bits que se obtiene aplicando una determinada función a un bloque de datos. El mensaje original se descompone en bloques, a cada uno se le aplica la función y el emisor envía el bloque y la suma de comprobación. El receptor aplica la misma función sobre cada bloque recibido y compara el resultado con la suma enviada.

Hay diferentes modos de definir la función, y, por tanto, varios algoritmos de detección de errores. Uno de los más sencillos es el de la **suma modular**:

- Cada bloque tiene m grupos de n bits cada uno.
- En cada bloque se suman los m grupos como números sin signo y descartando los bits de desbordamiento (es decir se hace una suma módulo 2^n) y se envía, junto con el bloque, el complemento a 2 del resultado.
- En el receptor se suman módulo 2^n los m grupos y el complemento a 2 recibido.
- El resultado tiene que ser 0. En caso contrario se ha producido un error en ese bloque.

Veamos un ejemplo con un mensaje de texto corto: «Hola ¿qué tal?» codificado en ISO-8859-15, con bloques de ocho grupos y ocho bits por grupo ($m = 8, n = 8$):

El primer bloque contiene las codificaciones de los ocho primeros caracteres, es decir, «Hola ¿qu» (Esto es porque estamos suponiendo un código ISO; si fuese UFT-8 tendría un carácter menos, porque «¿» se codificaría con dos bytes). El emisor envía los caracteres, y a medida que lo hace va calculando la suma modular de los grupos. Tras los ocho bytes, envía el complemento a 2 de la suma. El receptor, a medida que recibe bytes va realizando la suma, y tras recibir ocho bytes, hace el complemento a 2, y comprueba que es igual a 0. Concretamente:

Carácter	**ISO Latin-9 (hex)**	**ISO Latin-9 (bin)**		**En el receptor**
H	48	01001000	⇝	01001000
o	6F	01101111	⇝	01101111
l	6C	01101100	⇝	01101100
a	61	01100001	⇝	01100001
	20	00100000	⇝	00100000
¿	BF	10111111	⇝	10111111
q	71	01110001	⇝	01110001
u	75	01110101	⇝	01110101
Checksum:	(3)*49*	*01001001*		
Complemento a 2:		*10110111*	⇝	*10110111*
			Suma:	*00000000*

Si el resultado es 0, la probabilidad de que haya habido alteraciones en uno o varios bits es pequeña.

Para completar el ejemplo, el segundo bloque, que solo contiene seis caracteres («é tal?»), se completa con dos caracteres 0x00 («NUL»):

Carácter	ISO Latin-9 (hex)	ISO Latin-9 (bin)		En el receptor
é	E9	01100001	⇝	01100001
	20	00100000	⇝	00100000
t	74	10111111	⇝	10111111
a	61	01100001	⇝	01100001
l	6C	01101100	⇝	01101100
?	3F	00111111	⇝	00111111
NUL	00	00000000	⇝	00000000
NUL	00	00000000	⇝	00000000
Checksum:	(2)*89*	*10001001*		
Complemento a 2:		*01110111*	⇝	*01110111*
			Suma:	*00000000*

4.3. Control de errores

Detectado el error, es preciso corregirlo. Los dos enfoques principales son los que se conocen con las siglas «ARQ» y «FEC»:

Solicitud de repetición automática, ARQ (Automatic Repeat Request)

Los métodos de ARQ datan de los primeros tiempos de las comunicaciones de datos. Ya en el código ASCII se introdujeron dos caracteres de control para aplicarlos: ACK (acknowledge, 0x006) y NAK (negative acknowledge, 0x025).

Tras cada comprobación, el receptor envía un mensaje de reconocimiento positivo (si no ha detectado error) o negativo (si lo ha detectado). En el caso negativo, o si no recibe el mensaje tras un cierto tiempo *(«timeout»)*, el emisor retransmite las veces que sean necesarias.

La versión más sencilla es la de **parada y espera**: el emisor envía un bloque de bits y no envía el siguiente hasta no recibir respuesta del receptor; si ésta es positiva envía el bloque siguiente, y en caso contrario reenvía el mismo bloque. Requiere que sea posible transmitir en los dos sentidos, pero no simultáneamente: es lo que se llama «semidúplex».

Es relativamente fácil de implementar, pero el inconveniente de la parada y espera es su poca eficiencia. Dependiendo de la aplicación, los retrasos pueden ser inaceptables.

Mucho más eficientes, pero también más complejos, son los métodos de **transmisión contínua**: el emisor no espera, pero va analizando los mensajes de respuesta, y si alguno es negativo retransmite desde el bloque en el que se produjo el error. Esto implica dos cosas: por una parte, que la comunicación tiene que ser simultánea en los dos sentidos («full dúplex»), y por otra que los bloques tienen que ir numerados. Para esto último, en cada bloque, a los bits de datos y de comprobación se les añade una «cabecera» con un número de bloque en binario . Los algoritmos y protocolos concretos para la implementación de estos métodos se estudian en la asignatura «Teoría de la información».

Corrección de errores en destino, FEC (Forward Error Correction)

En algunas aplicaciones no es posible aplicar métodos de ARQ: aquellas en las que no hay un canal de retorno, o en las que los retardos son inaceptables. En esos casos se puede recurrir a métodos de FEC, que se basan en códigos correctores. Estos códigos introducen la redundancia suficiente para que el receptor pueda, no solo detectar un error en un bloque de bits, sino también recuperar los datos originales. Se utilizan, por ejemplo, en las transmisiones vía satélite y en los dispositivos de CD y DVD.

Los códigos correctores tienen una fundamentación matemática. Como decíamos antes, si en algunas asignaturas le enseñan temas que parecen (y son) abstractos, por ejemplo, modelos de Markov, sepa que éstos son la base de algoritmos como el de Viterbi, que permiten implementar la corrección de errores en la transmisión digital.

Finalmente, señalar que hay métodos que combinan las dos técnicas. Son los de **ARQ híbrido, HARQ (Hybrid ARQ)**: si el receptor no puede por sí solo recuperar los datos originales recurre a ARQ con un mensaje de reconocimiento negativo.

4.4. Compresión sin pérdidas *(lossless)*

La detección y corrección de errores es de aplicación en la transmisión de datos, ya sea a distancias más o menos largas (redes locales, Internet, comunicaciones por satélite...) o cortas (transferencias por buses, lectura y escritura en un DVD, copia de las fotos de una cámara a un disco...). La compresión de datos es asimismo de interés para la transmisión (enviar un flujo de datos por un medio con ancho de banda limitado), pero también para el almacenamiento (guardar un volumen de datos en menos espacio).

Por este motivo, si en los apartados anteriores hablábamos de «emisor-mensaje-receptor», en éste y el siguiente, para generalizar, diremos «codificador-datos (comprimidos) - descodificador». Cuando se trata de transmisión el codificador se encuentra en el emisor, los datos comprimidos forman el mensaje y el descodificador está en el receptor. En el caso de almacenamiento, el codificador es un programa que genera un fichero con los datos comprimidos, y el descodificador es otro programa que extrae los datos originales del fichero. Naturalmente, los algoritmos del codificador y del descodificador tienen que ser complementarios. Por eso, particularmente cuando se trata de datos multimedia, a la pareja se le llama **códec**.

Se dice que la compresión de datos es sin pérdidas cuando a partir de los datos comprimidos es posible recuperar exactamente los datos originales. Ya dijimos en la introducción de este capítulo que el principio básico de todos los algoritmos de compresión es reducir la redundancia. Hay muchos métodos para hacerlo, que además se suelen combinar.

Codificación de secuencias largas o RLE (Run-Length Encoding)

Si en la secuencia de símbolos que forman los datos originales hay subsecuencias de símbolos idénticos, en lugar de codificar uno a uno estos símbolos idénticos se puede codificar solo uno e indicar cuántas veces se repite. Como ilustración, el dato

`AAABBBBBBBBBCCBBFFFDDDDDDDDEEEEGGGGG`

se podría codificar así: `3A9B2C2B3F8D4E5G`
reduciéndose el número total de signos de 36 a 16.

En este ejemplo trivial, como $k = 36/16 = 2,25$, el factor de compresión es 2,25:1. En las aplicaciones reales las secuencias son mucho más largas y los factores de compresión mucho mayores.

Por ejemplo, en la asignatura «Introducción a la ingeniería» le explican que de la digitalización de una página para fax resultan, aproximadamente, 2.300×1.700 píxeles en blanco y negro, lo que requiere una capacidad de cerca de 4 Mbits. Suponga que la página tiene lo que muestra la figura 4.2: las franjas negras están separadas por 300 píxeles, y cada una tiene 50 píxeles de ancho. En lugar de representar cada línea mediante 1.700 bits podemos crear un código en el que «0» seguido de un número binario de 10 bits indique un número de ceros seguidos, y «1» seguido de otro número indique un número

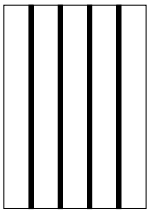

Figura 4.2.

de unos seguidos. Como hay cuatro franjas negras y cinco blancas, cada línea se representaría como una sucesión de $4 \times 11 + 5 \times 11 = 99$ bits, obteniéndose un factor de compresión de 1.700/99:1≈17:1. El código que se utiliza realmente para fax no es éste, pero el principio es el mismo.

Codificación con longitud variable, o codificación Huffman

El método de RLE está limitado porque no analiza los datos como un todo, solo tiene en cuenta secuencias, olvidando las anteriores. Así, en el último ejemplo está claro que no se aprovecha la redundancia de que todas las líneas son iguales.

Otro enfoque consiste en considerar todo el conjunto de datos y codificar los distintos símbolos con más o menos bits según que el símbolo aparezca con menos o más frecuencia. La idea es antigua: ya la aplicó intuitivamente Samuel Morse para su famoso código (circa 1850). El esquema de codificación y el algoritmo correspondiente más conocido es el que elaboró David Huffman en 1951 (siendo estudiante en el MIT, y como trabajo de una asignatura llamada «Teoría de la información»).

	N	Cod.
B	11	1
D	8	01
G	5	001
E	4	0001
F	3	00001
A	3	000001
C	2	0000001

Tabla 4.1. Ejemplo de código Huffman.

No presentaremos aquí el algoritmo, que se basa en unas estructuras llamadas «árboles binarios». Solamente un sencillo ejemplo para comprender su principio: la secuencia de 36 caracteres `AAABB...` que veíamos antes.

Si contamos el número de veces que aparece cada símbolo, podemos codificar el más frecuente como «1», el siguiente como «01», el siguiente como «001», etc., como indica la tabla 4.1.

Con este código, el dato original se codificaría como 000001000001.... El número total de bits es $11 \times 1 + 8 \times 2 + 5 \times 3 + 4 \times 4 + 3 \times 5 + 3 \times 6 + 2 \times 7 = 105$ en lugar de los $36 \times 8 = 288$ que resultan codificando todos los caracteres en ISO Latin9. Observe que, tal como se han elegido las codificaciones, no hay ambigüedad en la descodificación.

Como el código se elabora en el codificador de manera *ad-hoc* para unos datos originales concretos, quizás se esté usted preguntando «¿cómo sabe el descodificador cuál es el código?» Y la respuesta, efectivamente, es la que también estará pensando: los datos codificados tienen que ir acompañados del código. En este ejemplo trivial la tabla con el código ocuparía más bits que el mismo dato, pero en los casos reales los datos originales son muchos más voluminosos, y los datos del código son proporcionalmente mucho más pequeños.

Codificación por diccionario

En la codificación Huffman cada símbolo, o cada secuencia de bits de longitud fija, se codifica con un conjunto de bits de longitud variable. En la codificación por diccionario es al revés: se identifican secuencias de bits de longitud variable que se repiten y a cada una se le asigna una codificación de longitud fija. Por ejemplo en esta cadena:

`010-10100-1000001-10100-1000001-1000001-010-1000001-010-10100-010`

Secuencia	Índice	Cód.
010	I0	00
10100	I1	01
1000001	I2	10

Tabla 4.2 Ejemplo de diccionario.

podemos identificar secuencias que se repiten (los guiones solo están puestos para visualizarlas, no forman parte de los datos) y construir el «diccionario» de la tabla 4.2. Los datos codificados se construyen con las codificaciones sucesivas de los índices («entradas» del diccionario), en este caso, I0-I1-I2-I1-I2-I2-I0-I2-I0-I1-I0, que dan lugar a $2 \times 11 = 22$ bits en lugar de los 56 originales (en este ejemplo trivial el índice se codifica con solo dos bits).

En principio puede usted pensar que, como el diccionario se construye *ad-hoc* para los datos originales, sería necesario, como en la codificación Huffman, adjuntar este diccionario a los datos codificados para poder descodificarlos. Y así sería si el algoritmo fuese como sugiere el ejemplo. Pero el ejemplo está trivializado: el diccionario incluye también subsecuencias, y lo maravilloso del método es que el algoritmo que construye incrementalmente el diccionario conforme se van codificando los datos originales tiene una pareja: otro algoritmo que permite ir reconstruyendo el diccionario en el proceso de descodificación.

Los algoritmos más conocidos son LZ77 y LZ78 (por sus autores, Lempel y Ziv, y los años de sus publicaciones). De ellos han derivado otros, como el LZW, que es el que se utiliza para comprimir imágenes en el formato GIF, con el que se obtienen factores de compresión entre 4:1 y 10:1.

Deflate y otros

Deflate es un algoritmo complejo que, esencialmente, combina los métodos de LZ77 y Huffman. Se desarrolló en 1993 para el formato ZIP y el programa PKZIP (el nombre es por su autor, Phil Katz). Luego se adoptó oficialmente como estándar en Internet (RFC 1951, 1996) y es la base de dos formatos muy conocidos: gzip (genérico) y PNG (para imágenes). El factor de compresión de PNG es, en media, entre un 10 % y un 30 % mejor que el de GIF.

Se han elaborado otros algoritmos derivados de LZ77, y algunos han dado lugar a productos patentados. Uno de los más conocidos es RAR (Roshal ARchive), desarrollado por el ruso Eugene Roshal en 1993 y muy utilizado en entornos Windows.

Hay otros métodos que utilizan varias de las técnicas anteriores y otras que no hemos mencionado. Por ejemplo, bzip2 y 7-Zip, muy eficientes en factor de compresión, pero que exigen más recursos (fundamentalmente, tiempo de ejecución del algoritmo de compresión). 7-Zip es, realmente, un formato con arquitectura abierta que permite acomodar distintos métodos de compresión: Deflate, bzip2, etc.

Compresores y archivadores

En los párrafos anteriores hemos aludido a algunos algoritmos de compresión (RLE, Huffman, LZ77, LZ78, LZW, Deflate) y algunos formatos de ficheros para imágenes (GIF, PNG) o genéricos (ZIP, gzip, RAR, bzip2, 7-Zip) en los que se aplican esos algoritmos. Ahora bien, entre estos últimos hay dos tipos: gzip y bzip2 son, simplemente, *compresores*. ZIP, RAR y 7-Zip son también *archivadores*.

Aquí conviene explicar la diferencia entre **fichero** *(file)* y **archivo** *(archive)*. Son cosas distintas que suelen confundirse debido al empeño de cierta empresa dominante en Informática de traducir «file» por «archivo»[1].

Un fichero es un conjunto de datos codificados en binario (comprimidos o no), almacenados en un soporte no volátil con un nombre asociado, y disponible para los programas que pueden interpretar esos datos. (Ese conjunto de datos puede ser, en sí mismo, un programa).

Un archivo es un fichero que contiene varios ficheros junto con datos sobre estos ficheros («metadatos», apartado 5.1) que permiten extraer los ficheros mediante un programa adecuado.

Pues bien, gzip y bzip2 solo son compresores: su entrada es un fichero y su salida es otro fichero comprimido. ZIP, RAR y 7-Zip son, también, archivadores: pueden recibir como entrada varios ficheros y generar un archivo comprimido. Además, incluyen, opcionalmente, una función que no hemos tratado en este Tema: el cifrado (a veces llamado «encriptación»).

No es causalidad que gzip y bzip2 tengan su origen en el «mundo Unix», mientras que los otros proceden del «mundo Windows». Uno de los principios de la «filosofía Unix» es que los programas

[1] Como tampoco debería traducirse «directory» por «carpeta»: ésta sería la traducción de «folder».

deben hacer una sola cosa (pero hacerla bien) y otro, que deben diseñarse de modo que se puedan encadenar fácilmente. Los programas archivadores en Unix son «`ar`», «`shar`» y el más utilizado, «`tar`». Por ejemplo, para crear un archivo comprimido de todos los ficheros contenidos en el directorio `dir/` se puede utilizar esta orden para el intérprete:

```
tar cf - dir/ | bzip2 > dircompr.tar.bz2
```

- «`c`» significa «archivar» (para extraer, «`x`», y para listar, «`t`»).
- «`f`» significa que a continuación se da el nombre de un fichero para el resultado.
- «`-`» significa que el resultado no vaya a un fichero, sino a la «salida estándar».
- «`dir/`» es el directorio que contiene los ficheros a comprimir (aquí podrían ponerse varios ficheros y/o directorios).
- «`|`» hace una «tubería» (*pipe*): la salida estándar de lo que hay atrás se utiliza como entrada para lo que sigue.
- «`bzip2`» comprime lo que le llega y envía el resultado a la salida estándar.
- Como ya sabe usted por las prácticas del Tema 1, «`>`» hace una *redirección*: dirije la salida estándar hacia el fichero que hemos llamado `dircompr.tar.bz2`.

(El programa `tar` incluye una facilidad con la que se puede hacer lo mismo escribiendo menos: `tar cjf - dir/ > dircomp.tar.bz2`. El argumento «`j`» hace que automáticamente se cree la tubería y se ejecute `bzip2`. Con «`z`» se ejecutaría `gzip`).

4.5. Compresión con pérdidas *(lossy)*

En la compresión con pérdidas se sacrifica el requisito de que a partir de los datos comprimidos se puedan reconstruir *exactamente* los originales. Típicamente se aplica a datos audiovisuales (o sensoriales, en general), tanto para su almacenamiento como para su transmisión, porque el agente destinatario final de los datos, normalmente humano, es capaz de recuperar la información siempre que la distorsión producida por la pérdida de datos no sea muy grande.

La elección de un algoritmo de compresión es muy dependiente de la aplicación y de los recursos disponibles. En general (sea con pérdidas o sin ellas) es necesario un equilibro entre

- Factor de compresión: cuanto mayor sea, menos memoria se requiere para almacenar los datos, y menos ancho de banda para su transmisión.
- Recursos disponibles: podemos tener un algoritmo muy eficiente (que consiga un factor de compresión muy grande) pero que no sirva porque la memoria disponible para el programa sea insuficiente, o el tiempo necesario para ejecutarlo sea inaceptable.

Con los algoritmos de compresión con pérdidas se consiguen factores de compresión mayores que sin pérdidas, pero aparece un tercer elemento a considerar:

- Grado de distorsión aceptable.

El problema es que este grado es subjetivo. Un ejemplo que se entenderá fácilmente es el de la música codificada en MP3. Para la mayoría de las personas, con el factor de compresión habitual en MP3 (11:1, a 128 kbps), el resultado es más que aceptable. Pero para ciertos oídos exquisitos no (normalmente son los mismos a los que les suena mejor la música grabada en vinilo que en CD).

Ahora bien, como usted ya sabe, los datos procedentes de las señales acústicas y visuales se obtienen mediante un proceso de muestreo y cuantificación, y en este sentido ya han sufrido «pérdidas». Si se reduce la frecuencia de muestreo, o el número de niveles de cuantificación, para una misma señal se generan menos datos, pero la reconstrucción de la señal es menos fiel. En principio, la «compresión» se refiere al proceso posterior, en el que se aplica un algoritmo sobre los datos ya digitalizados. Sin embargo, aunque teóricamente la diferencia entre los dos procesos está clara, en la práctica es difícil separarlos, porque ambos se combinan para optimizar el resultado. Es por eso que se utilizan los términos más generales «codificación» y «descodificación», entendiéndose que engloban tanto la discretización (conversión de analógico a digital) y la reconstrucción (conversión de digital a analógico) como la compresión y la descompresión. Un **códec**, como ya hemos dicho, es una pareja de codificador y descodificador diseñados de acuerdo con unos convenios sobre discretización y algoritmo de compresión.

Aunque hay muchos tipos de códecs para datos de audio, de imágenes y de vídeo, casi todos comparten dos principios de diseño:

- Se basan en **codificación perceptual**, es decir, el esquema de codificación se diseña teniendo en cuenta características fisiológicas de la percepción. Normalmente, esto implica una transformación del dominio temporal de las señales originales a un dominio en el que se expresa mejor la información que contienen. Por ejemplo, en lugar de representar los sonidos como una sucesión de valores de amplitud en el tiempo, se transforman para representarlos como variaciones en el tiempo de componentes de frecuencias, lo que se aproxima más a los modelos de la percepción humana.

- Esas transformaciones de dominios de representación están perfectamente estudiadas matemáticamente, y las estudiará usted en la asignatura «Señales y sistemas». La más utilizada en los algoritmos de compresión con pérdidas es la **DCT** (transformada discreta del coseno), que es una variante de la más clásica transformada de Fourier.

Para concretar, aunque sin entrar en los algoritmos, veamos cómo se aprovechan algunas de esas propiedades de la percepción de distintos tipos de medios.

Sonidos

El rango de frecuencias audibles es, aproximadamente, de 20 Hz a 20.000 Hz, pero para aplicaciones en las que solo interesa que la voz sea inteligible (como la telefonía) basta un rango de 300 Hz a 3.500 Hz. Como se explica en la asignatura «Introducción a la ingeniería», el teorema del muestreo de Nyquist-Shannon permite calcular la frecuencia de muestreo mínima para poder reconstruir la señal. Reduciendo esta frecuencia reducimos el número de bits necesario para codificar la señal, a costa de perder calidad del sonido.

Esta característica de la percepción se aplica al proceso de discretización, pero hay otras que sirven también para la compresión. Por ejemplo, el *enmascaramiento:* dos sonidos de frecuencias próximas que se perciben bien separadamente, pero solo se percibe el más intenso cuando se producen conjuntamente (enmascaramiento simultáneo) o cuando uno precede a otro (enmascaramiento temporal).

Todo esto se tiene en cuenta para diseñar procedimientos que facilitan la compresión. Por ejemplo, SBC *(Sub-Band Coding)* consiste en descomponer la señal en bandas de frecuencias y codificar cada banda con un número diferente de bits. Forma parte de los algoritmos utilizados en muchos formatos de audio: MP3, AAC, Vorbis, etc.

Hay características que son específicas de la voz humana, no de la percepción, y en ellas se basan algoritmos optimizados para aplicaciones de voz, como CELP (Code Excited Linear Prediction), o AMR (Adaptive Multi-Rate audio codec), que se utiliza en telefonía móvil (GSM y UMTS).

Imágenes

Una propiedad importante de la percepción visual es que el ojo humano es menos sensible a las variaciones de color que a las de brillo.

Para conseguir una representación «perfecta» del color *(«truecolor»)*, en cada píxel se codifican los tres colores básicos (rojo, verde y azul) con ocho bits cada uno (256 niveles para cada color). Es decir, se utilizan 24 bits (tres bytes) por píxel. Para almacenar una imagen de, por ejemplo, 2.000×2.600 píxeles se necesitan $3 \times 2.000 \times 2.600/2^{20} \approx 143$ MiB. Obviamente, se puede reducir el tamaño reduciendo la resolución (el número de píxeles), pero también reduciendo los bits necesarios para codificar el color.

En el caso de dibujos en los que no es necesario reproducir un número tan elevado de colores como en la fotografía se utiliza un solo byte para el color. Es común en aplicaciones de dibujo tener una **paleta de colores**: un conjunto de 256 colores que se pueden seleccionar previamente. Cada píxel se codifica con un solo byte, que es un índice a la paleta. Es lo que se hace en el formato GIF (recuerde que en este formato, después de esta discretización, se aplica un algoritmo de compresión sin pérdidas, el LZW).

Para imágenes reales 256 colores son insuficientes, y se aplica un procedimiento más sutil para aprovechar esa propiedad de menor sensibilidad a las variaciones de color («croma») que a las de brillo («luma»). En la representación de un píxel por las intensidades de sus tres colores básicos se mezclan la información de color y la de brillo. Se puede hacer una transformación matemática del espacio de representación RGB (rojo, verde, azul) en un espacio en el que una dimensión corresponda al brillo y las otras dos al color. Hay varias maneras de definir esa transformación. En televisión analógica, las normas NTSC definían el espacio llamado «YIQ», aunque más tarde adoptaron el de PAL, «YUV». En codificación digital se utiliza otro espacio, el «**YCbCr**». Esencialmente, «Y» es el luma, un valor proporcional a R+G+B, y «Cb» y «Cr» proporcionan el croma: son proporcionales a B–Y y R–Y, respectivamente.

La mayoría de los algoritmos de compresión de imágenes y de vídeos (JPEG, MPEG, DivX, Theora, etc.) se apoyan en esta transformación para aplicar el método de **submuestreo del color** *(chroma subsampling)*. Consiste, simplemente, en utilizar más muestras para Y que para Cb y Cr. Hay una notación para expresar la proporción de muestras de cada componente: «(4:*a*:*b*)» significa que en una región de cuatro píxeles de ancho y dos de alto se toman *a* muestras de (Cb,Cr) en la primera línea y *b* en la segunda. En el estándar DV (Digital Video) se usa (4:1:1); en JPEG, en MPEG y en DVD, (4:2:0).

Otro procedimiento común en los algoritmos de compresión es el **muestreo de altas frecuencias**. Se aplica a los coeficientes que resultan de aplicar la DCT: los correspondientes a frecuencias altas se codifican con menos bits.

Los programas que aplican estos algoritmos suelen tener una opción para definir un parámetro de calidad, entre 0 y 100. Cuanto menor es el factor mayor es el factor de compresión (y mayores las pérdidas perceptibles).

Imágenes en movimiento

La propiedad más importante de la visión humana aprovechable para la codificación de imágenes en movimiento es la de **persistencia**: la percepción de cada imagen persiste durante, aproximadamente, 1/25 seg. Esto conduce a lo que podemos llamar el *«principio de los hermanos Lumière»*: para conseguir la ilusión de movimiento contínuo basta con presentar las imágenes sucesivas a un ritmo de 30 fps. «fps» es la abreviatura de «frames por segundo»; en este contexto, una traducción adecuada de «frame» es «fotograma».

En animaciones en las que se puede admitir una pequeña percepción de discontinuidad la tasa puede bajar a 12 fps. En cinematografía son 24 fps, pero duplicados o triplicados ópticamente (la película avanza a 24 fps, pero con esa tasa se percibiría un parpadeo; el proyector repite dos o tres veces cada fotograma). En televisión analógica los estándares son 25 fps (PAL) o 30 fps (NTSC), pero en realidad se transmiten 50 o 60 campos por segundo entrelazados (un campo contiene las líneas pares y otro las impares). La HDTV en Europa utiliza 50 fps. Los monitores LCD suelen configurarse para 60 o 75 fps.

Aunque los fotogramas estén comprimidos, basta hacer unos sencillos cálculos para comprobar que con esas tasas de fotogramas resultan unas tasas de bits que exceden la capacidad de los medios de transmisión normales y harían imposible, por ejemplo, el «streaming» por Internet. Pero a esa *compresión intrafotograma*, que aprovecha la *redundancia espacial* se le puede añadir una *compresión interfotograma*, que aprovecha la *redundancia temporal*. Esta redundancia procede, por una parte, de otra propiedad fisiológica: el sistema visual no aprecia distorsiones si en la secuencia de fotogramas algunos no son «reales», sino «promedios» de los próximos. Por otra, es muy frecuente que fotogramas sucesivos solo difieran en algunos detalles (por ejemplo, en una secuencia en la que un objeto se mueve, pero el fondo de la imagen permanece prácticamente constante).

La duplicación óptica del cine aprovecha esa redundancia temporal, pero en los códecs se utiliza de manera más elaborada. En todos los estándares MPEG se aplica lo que se llama **compensación de movimiento**, que, resumidamente, consiste en lo siguiente:

- Una secuencia de fotogramas se estructura en grupos. Cada grupo se llama «GOP» (group of pictures). No todos los fotogramas de un GOP se codifican (comprimidos) con toda la resolución, solo algunos.
- Los «I-frames» (intra) se comprimen normalmente con algoritmos similares a JPEG.
- Los «P-frames» (predichos) contienen solamente las diferencias con el fotograma anterior y se comprimen mucho más.
- Los «B-frames» (bi-predichos) se obtienen del anterior y del siguiente y se comprimen aún más.

Capítulo 5

Almacenamiento en ficheros

En el Tema 1 se introdujo el concepto de fichero y en éste hemos vuelto sobre él para matizar la diferencia entre «fichero» y «archivo», que es un tipo de fichero (apartado 4.4). También se explicó que el sistema de gestión de ficheros (SGF), que forma parte del sistema operativo, contiene programas para acceder a los datos de los ficheros, almacenados en los dispositivos de memoria secundaria (disco magnético, CD, DVD, *flash*, etc.).

La transferencia de datos entre las memorias secundarias y la memoria principal (de acceso aleatorio) se realiza mediante la técnica de acceso directo a memoria (DMA, apartado 1.3): en cada operación de lectura o escritura en la memoria secundaria se transfiere un **bloque**: es un múltiplo de 512 bytes formado por uno o varios **sectores**, que es como normalmente se organizan los datos en el soporte físico. El SGF se ocupa de la correspondencia entre los sectores y los ficheros, asignando sectores a ficheros cuando los necesitan, o liberándolos cuando ya no son necesarios. Por ejemplo, cuando se borra un fichero, realmente no se borran físicamente los datos de los sectores, simplemente se liberan esos sectores (por eso hay herramientas que permiten recuperar ficheros borrados).

Una cosa es el SGF, conjunto de programas que hacen todo eso, y otra el **sistema de ficheros**, que es la forma de organización de los ficheros, directorios y dispositivos periféricos. Todos los sistemas operativos tienen un SGF que trata con un sistema de ficheros, pero es frecuente que pueda tratar varios. Por ejemplo, en los sistemas Windows actuales el sistema de ficheros se llama «NTFS», pero el SGF puede tratar también el más antiguo «FAT32», que aún se usa, por ejemplo, en memorias *flash* extraíbles. En Linux, los sistemas de ficheros principales son «ext3», o el más reciente «ext4» (que es el que hemos utilizado en las prácticas del Tema 1), pero el SGF puede tratar otros, como FAT32 y NTFS. Además, hay sistemas de ficheros especializados para ciertos soportes, como el «ISO 9660» para CD y DVD.

En este capítulo presentaremos algunas características de los ficheros que son comunes e independientes del sistema de ficheros. Pero veamos antes los distintos tipos de ficheros.

5.1. Tipos de ficheros

Ficheros especiales

Todos los sistemas operativos modernos han adoptado una idea procedente de Unix: integrar en el SGF los dispositivos de entrada y salida. En Unix se agrupan en el directorio `/dev`. El objetivo es hacer uniforme el acceso mediante llamadas estándar al sistema. Hay *ficheros especiales de caracte-*

res y *ficheros especiales de bloques*. Los primeros corresponden a periféricos lentos, como el teclado (`/dev/tty`) o el ratón (`/dev/input/mouse`), y los segundos a los rápidos, que transfieren mediante DMA, como los discos magnéticos (`/dev/hda`, o `/dev/sda`) o los ópticos (`/dev/cdrw` o `/dev/dvdrw`) (independientemente de que dentro de éstos haya ficheros «normales», o «regulares»).

También hay ficheros especiales con funciones específicas llamados «pseudodispositivos». Un ejemplo es `/dev/null`, que sirve de «sumidero»: todo lo que se envía a él se pierde. Otros ficheros especiales son los directorios. Y hay algunos más, como los enlaces simbólicos y los *sockets*.

Ficheros regulares

En el resto de este capítulo trataremos únicamente de los ficheros que se llaman «regulares»: los que contienen datos almacenados en algún soporte de memoria secundaria. La abstracción que ofrece el sistema de ficheros permite que los programas que manipulan esos ficheros los «vean» simplemente como una secuencia de bytes, sin ocuparse de cómo se almacena esa secuencia de bytes en el soporte. Por ejemplo, en un disco magnético un fichero se almacena en «sectores» (normalmente, de 512 bytes) que están distribuidos por las distintas pistas del disco, pero para los programas que hacen llamadas al SGF se abstraen esos detalles de localización.

Por otra parte, en la mayoría de los sistemas operativos el sistema de ficheros no se ocupa del contenido de los ficheros: para él, un fichero no es más que una secuencia de bytes. Es decir, solamente ofrece un tipo de fichero regular. La estructura y la interpretación del contenido de un fichero queda a cargo de los programas de aplicación que lo usan.

Metadatos

Para realizar las operaciones con los ficheros (crear, borrar, modificar, etc.), el SGF tiene que poder identificar a cada fichero por su nombre y conocer sus propiedades: tipo (especial, directorio, regular, etc.), fechas de creación y de última modificación, propietario, permisos de lectura, escritura y ejecución, y si es regular, bloques que tiene asignados, tamaño (en bloques o en bytes), etc. Algunos de estos datos son los que ha visto usted en las prácticas del Tema 1 con la orden «`ls -l`». Como son datos sobre los datos, se llaman **metadatos**. Pero observe que estos metadatos se refieren a propiedades generales del fichero, no a su contenido. También hay metadatos sobre el contenido, como veremos el siguiente apartado.

La forma de mantener los metadatos es distinta para cada sistema de ficheros. El sistema «FAT» debe su nombre a que guarda los de todos los ficheros en una estructura de datos llamada «tabla de asignación de ficheros». En NTFS es la «MFT» (Master File Table). El enfoque de Unix es diferente: cada fichero tiene asociada una pequeña estructura (normalmente, 256 bytes), llamada «**inode**» con sus metadatos. (Se pueden ver los números de los inodes asociados a los ficheros con «`ls -i`»).

Pero veamos los convenios para los tipos de ficheros regulares, independientemente de los sistemas de ficheros.

5.2. Ficheros regulares

Identificación del tipo de contenido

Para el sistema de ficheros todos los ficheros regulares son iguales, pero para los programas que trabajan con ellos es necesario conocer lo que contienen: texto, una imagen, un código ejecutable... Hay varias maneras de hacerlo.

Extensión del nombre

Una forma muy sencilla de identificar el tipo de contenido es indicarlo en el mismo nombre, con el convenio de que este nombre termine siempre en un punto seguido de varios caracteres que se llaman «**extensión**» y que definen el tipo. El sistema DOS y los primeros Windows tenían la limitación de que la extensión no podía tener más de tres caracteres, pero esta limitación ya no existe. Por ejemplo, los documentos en HTML tienen un nombre terminado en `.htm` o `.html`, las imágenes en JPEG, `.jpg` o `.jpeg`, los programas fuente en lenguaje C, `.c`, en Java, `.java`, etc.

Metadatos internos

Un método más seguro es incluir los datos sobre el tipo de contenido dentro del mismo fichero, en una posición predeterminada, normalmente al principio y antes de los datos propiamente dichos, formando la **cabecera** *(header)*.

La cabecera puede ser más o menos larga y compleja, desde los ficheros de texto, en los que no existe (salvo en UTF-16 y UTF-32, que pueden llevar dos bytes al principio con el BOM, apartado 2.5), hasta los ficheros de multimedia, que normalmente tienen muchos metadatos. En cualquier caso, la cabecera, si existe, empieza con unos pocos bytes que identifican el tipo de contenido, y se les llama «número mágico».

Números mágicos y el programa `file`

Algunos ejemplos de números mágicos:

- Los ficheros que contienen imágenes GIF empiezan con un número mágico de seis bytes que son las codificaciones en ASCII de «GIF89a» o de «GIF87a». A continuación, dos bytes con el número de píxeles horizontales y otros dos con el número de píxeles verticales.
- Los que contienen imágenes JPEG empiezan todos con 0xFFD8 y terminan con 0xFFD9 (al final del fichero). Pueden contener más metadatos, según las variantes. Por ejemplo, con el formato «Exif» *(Exchangeable image file format)*, común en las cámaras fotográficas, los metadatos (fecha y hora, datos de la cámara, geolocalización, etc.) forman ya una cabecera relativamente grande.
- Los PNG empiezan con ocho bytes: 0x89504E470D0A1A0A.
- Los PDF, con « %PDF» (0x25504446).
- Los ficheros de clases Java compiladas (.class), con 0xCAFEBABE
- Los archivos comprimidos ZIP, con «PK» (0x504E).
- Los ficheros ejecutables de Windows, con «MZ» (0x4D5A).
- Los ficheros ejecutables de Unix, con «.ELF» (0x7F454C46).
- Los ficheros ejecutables de Mac OS X («Mach-O») pueden empezar con 0xFEEDFACE (PowerPC), con 0xCEFAEDFE (Intel) o con 0xCAFEBABE (Universal, ver apartado 5.5).
- A veces el número mágico no está en los primeros bytes. En los ficheros de archivo `tar` son los caracteres «ustar» a partir del byte 0x101 de la cabecera.

«`file`» es un programa de utilidad para sistemas Unix que identifica el tipo de contenido de un fichero. En principio, se basa en comparar los primeros bytes del fichero con los números mágicos guardados en un fichero de nombre «`magic`» (su localización depende del sistema: puede estar en `/etc/magic`, o en `/usr/share/file/magic`, o en otro directorio). En realidad, hace tres pruebas:

1. Prueba con los metadatos del SGF (contenidos en el inode) para ver si es un fichero especial: directorio, dispositivo, etc. Si es así informa de ello y si no, es que es un fichero regular y pasa a la segunda prueba
2. Prueba con los números mágicos de `magic`. Si encuentra coincidencia informa y, dependiendo de lo que ha identificado, continúa leyendo bytes de la cabecera para dar más información. Por ejemplo, para un GIF mira las dimensiones. Si no logra identificar ningún número mágico pasa a la tercera prueba.
3. Prueba de lenguaje. Como los ficheros de texto plano no tienen cabecera ni número mágico, el programa examina la secuencia de los primeros bytes. Normalmente acierta, identificándolo como ASCII, ISO Latin1, UTF-8, etc. Otras pruebas de naturaleza más heurística le permiten identificar tipos particulares de ficheros de texto. Por ejemplo, los documentos HTML tienen al principio «`html`» o «`HTML`», los programas fuente del lenguaje Java normalmente contienen declaraciones como «`public class...`» y sentencias como «`import java.util.*;`», etc.

En cualquier caso, el programa `file` no es infalible, y es fácil «engañarlo». Un ejercicio interesante que puede usted hacer (y así repasa la representación de caracteres y de números): escriba un fichero de texto plano con los caracteres ASCII «GIF89aABCD», guárdelo con un nombre cualquiera (por ejemplo, «`ppp`») y déselo a `file`. Obtendrá esto:

```
$ file ppp
ppp: GIF image data, version 89a, 16961 x 17475
```

Habiendo identificado el número mágico, ha seguido mirando bytes y ha interpretado los siguientes como las dimensiones. El ejercicio consiste en comprobar que `file` ha interpretado las codificaciones ASCII de «ABCD» como dos números de 16 bits con el convenio extremista menor y de ahí han resultado esas «dimensiones».

Metadatos externos y tipos MIME

Ya hemos dicho que normalmente los sistemas operativos no mantienen metadatos sobre el contenido de los ficheros en estructuras del SGF, que serían metadatos externos al propio fichero. Pero hay otra forma de codificar metadatos que está relacionada con los protocolos de comunicación. Aunque raramente se utiliza para el almacenamiento en ficheros, que es de lo que estamos tratando, tiene que ver con la identificación del tipo de fichero y es interesante conocerla. Veamos brevemente de qué se trata.

Se trata de MIME (Multipurpose Internet Mail Extensions). En su origen se definió como un modo de extender el correo electrónico (que inicialmente era solo para texto ASCII) con textos no ASCII, imágenes, sonidos, etc. Se usa en los protocolos de correo (SMTP), y también en los de la web (HTTP) y en otros protocolos.

Es un sistema de identificadores estandarizado por la IANA (Internet Assigned Numbers Authority). Cada identificador consta de un «tipo» y un «subtipo» separados por el símbolo «/» En la cabecera que precede al fichero que contiene los datos se indica el tipo y el subtipo: text/plain, text/html... audio/mp4, audio/ogg... video/mpeg, video/quicktime...

De este modo, el receptor (el lector de correo, o el navegador) sabe qué programa tiene que utilizar para interpretar los datos, sin necesidad de fiarse de la extensión ni de abrir el fichero para ver su cabecera.

La IANA tiene definidos nueve tipos (application, audio, example, image, message, model, multipart, text y video) y para cada uno, muchos subtipos.

Formatos de ficheros

En el capítulo 3 hemos estudiado formatos para tipos de datos simples, como son los números, que se representan con pocos bits: una o varias palabras. Ahora hablamos de datos que pueden ocupar un volumen grande, de MiB o GiB, y como estos bits pueden representar datos de naturaleza muy diversa, hay una gran variedad de formatos.

Los detalles de cada formato son muy prolijos y solo tiene sentido estudiarlos si se va a trabajar, por ejemplo, en el diseño o en la implementación de un procesador de textos o de un códec. En los apartados siguientes veremos algunos aspectos generales de los formatos más conocidos para los distintos tipos de ficheros.

5.3. Ficheros de texto, archivos, documentos y datos estructurados

Hay que distinguir entre texto plano y texto con formato. En el primer caso, los datos están formados por la cadena de bytes correspondiente a las codificaciones de los caracteres (apartado 2.5), y el contenido del fichero no es más que esa secuencia de bytes, sin más formato. Solamente varían pequeños detalles de un sistema a otro. Por ejemplo, la manera de señalar el final de una línea. En Windows se hace con una combinación de dos caracteres de control: retorno (CR, 0x0D) y nueva línea (LF, 0x0A), mientras que en Unix y en Mac OS X es simplemente LF. En algunos sistemas primitivos se señalaba el fin del texto mediante otro carácter de control: «Control-Z» (0x1A). Ya no es necesario (aunque aún se encuentra en algunos ficheros de texto), puesto que la longitud en bytes del fichero siempre está incluida en los metadatos del SGF.

La semántica del texto depende del agente que lo lee o lo escribe. Puede ser muchas cosas: un mensaje para ser leído por personas, unos datos de configuración de una aplicación, un programa escrito en un lenguaje de programación, una página web escrita en HTML, una lista de teléfonos...

Ficheros de archivos

El formato de un archivo «puro», es decir, sin comprimir, es muy sencillo: es una concatenación de los ficheros, con algunos metadatos. Por ejemplo, en un fichero (archivo) `tar` cada uno de los ficheros que lo componen va precedido de una cabecera con el nombre y los demás metadatos (propietario, permisos, fechas, etc.). Normalmente el archivo se comprime, como hemos visto en un ejemplo en el apartado 4.4, formando lo que se llama un «**tarball**». Un fichero comprimido tiene también su cabecera antes de los datos comprimidos. En `gzip` son diez bytes con el número mágico, versión y fecha, y, tras los datos, ocho bytes para comprobación de errores con CRC-32 (apartado 4.1).

El formato de los ficheros ZIP, al tratarse de una combinación de archivo y compresión y, opcionalmente, cifrado, es bastante más complejo.

Ficheros de documentos

Para los textos «enriquecidos» con propiedades de los caracteres (tamaño, tipo de letra o *«font»*, color, etc.) y para otros documentos que pueden incluir también imágenes y sonidos (presentaciones, hojas de cálculo, etc.), durante muchos años se han ido sucediendo «estándares *de facto*»: convenios para programas comerciales que durante algún tiempo se han impuesto en el mercado de la ofimática. Los ejemplos más recientes son los de la «suite» Microsoft Office, que incluye el programa de procesamiento de textos «Word» y su formato «.doc», el de presentaciones «PowerPoint» y su formato

«.ppt» el de hojas de cálculo «Excel» y su formato «.xls», etc. Estos formatos son privativos («propietarios»), ligados a los programas comerciales que los usan. Recientemente, debido en parte a la presión de las administraciones europeas por el uso de formatos abiertos, se están extendiendo dos estándares oficiales:

- **ODF** (Open Document Format) es un estándar ISO (ISO/IEC 26300:2006) que incluye convenios para textos (ficheros «`.odt`»), presentaciones («`.odp`»), hojas de cálculo («`.ods`»), etc.
- **OOXML** (Office Open XML) es otro estándar (ISO/IEC IS 29500:2008) similar, promocionado por Microsoft. El convenio para los nombres de los ficheros es añadir la letra «x» a las extensiones de los antiguos formatos: «`.docx`», «`.pptx`», «`.xlsx`», etc.

Ambos comparten el basarse en un metalenguaje que estudiaremos en el Tema 3, XML. Todo documento se representa mediante un conjunto de ficheros XML archivados y comprimidos con ZIP. Puede usted comprobarlo fácilmente: descomprima un documento cualquiera que esté en uno de estos formatos (aunque no tenga la extensión «.zip») y verá aparecer varios directorios («carpetas») y en cada directorio varios ficheros XML, ficheros de texto plano que puede usted leer (con `cat`, o con `more`, o con `less`...) y editar con cualquier editor de textos.

PDF *(Portable Document Format)* es otro formato muy conocido para distribución de documentos. En su origen también privativo, es actualmente un estándar abierto (ISO 32000-1:2008), y la empresa propietaria de las patentes (Adobe) permite el uso gratuito de las mismas, por lo que hay muchas implementaciones tanto comerciales como libres. Es un formato complejo, muy rico en tipografías y elementos gráficos matriciales y vectoriales.

Ficheros de datos estructurados

Muchas aplicaciones de la Informática (de hecho, las más «tradicionales») tienen que ver con conjuntos de objetos o individuos, cada uno con sus propiedades, o atributos. Los datos correspondientes a esos atributos se reparten en distintos ficheros, y cada fichero contiene un conjunto de valores de atributos para un tipo de objeto determinado.

Por ejemplo, una universidad tiene que mantener datos sobre alumnos, centros, titulaciones, proyectos, etc. Uno de los ficheros puede ser el de centros. Para cada centro tendrá registrado el nombre, las titulaciones que ofrece, las asignaturas que imparte, etc. Los profesores pueden estar en otro fichero, y para cada profesor tendrá también sus propiedades.

Un fichero de este tipo contiene, para cada uno de los objetos o individuos, un conjunto de datos que se llama **registro** *(«record»)*[1]. Los registros deben tener una determinada forma, o **esquema**: definición de las partes o **campos** que lo forman. Cada campo corresponde a un atributo, y el esquema define el **tipo** de ese atributo (valores posibles que puede tomar). Por ejemplo, el esquema puede definir el campo «Nombre» de tipo *«string»* (cadena de caracteres), el campo «Fecha de nacimiento» de tipo *«date»*, etc.

La aplicación final deberá facilitar el acceso controlado a esos datos para los distintos tipos de usuarios. Por ejemplo, el administrador podrá insertar, modificar y borrar registros, mientras que un alumno no podrá hacerlo, pero sí consultar sus calificaciones.

La implementación de estas aplicaciones en «bajo nivel», es decir, programando los detalles de cómo se accede a un campo de un registro, cómo se busca, etc., teniendo en cuenta que se trata de

[1] Aunque en español los llamemos igual, estos registros son estructuras de datos que no tienen nada que ver con los registros hardware *(«registers»)*, componentes físicos de almacenamiento (apartado 1.3).

multiples ficheros interrelacionados, sería muy laboriosa. Existen herramientas genéricas que facilitan su diseño, su construcción y su uso. Son los **sistemas de gestión de bases de datos**, cuyos fundamentos estudiaremos en el Tema 5.

5.4. Ficheros multimedia

Los ficheros multimedia pueden almacenar una variedad de tipos de datos, y generalmente se agrupan en un fichero compuesto llamado «contenedor». Veamos separadamente los distintos tipos de ficheros que, además de los de texto y documentos, pueden agruparse en un contenedor.

Ficheros de sonidos

En su función de codificación, un códec de audio aplica (mediante software o mediante hardware) determinados algoritmos al resultado de la conversión analógica-digital (que se explica en la asignatura «Introducción a la Ingeniería») y genera un resultado que puede transmitirse por un canal de comunicación *(streaming)* o almacenarse en un fichero siguiendo los convenios de un formato. Los algoritmos y los formatos están íntimamente relacionados, y dependen de la aplicación. Los podemos clasificar según el grado de compresión (apartados 4.4 y 4.5) que aplican:

- Sin compresión. La mayoría están basados en PCM *(Pulse Code Modulation)*: los valores codificados de las muestras son proporcionales a la amplitud o a su logaritmo. De este tipo es el formato estándar de Audio-CD, así como WAV (Microsoft/IBM), AIFF (Apple) y AU (Sun). El primero establece unos parámetros fijos (2 canales muestreados a 44.100 Hz y cuantificados con 16 bits por muestra). Los códecs de los otros permiten seleccionar los parámetros para adaptar el resultado a diferentes necesidades (ancho de banda, capacidad de almacenamiento, calidad...).
- Con compresión sin pérdidas. Pueden llegar a un porcentaje de compresión de 60-50 % (entre 1,66:1 y 2:1), y, como se puede recuperar íntegramente el original, son muy adecuados para archivos sonoros. Un ejemplo es FLAC *(Free Lossless Audio Codec)*, que utiliza para la compresión RLE y predicción lineal. Otro, WMA Lossless, solo para sistemas Windows (no está documentado).
- Con compresión con pérdidas. El formato más conocido actualmente es el de MP3 (MPEG-1 Audio Layer 3), que se define como una secuencia de «marcos» *(frames)*. Cada marco tiene una cabecera con bits de sincronización y metadatos: la frecuencia de muestreo, la tasa de bits *(bitrate)*, las etiquetas ID3, etc., seguida de datos. Las etiquetas ID3 codifican metadatos como título, artista, número de pista, etc.

Otros formatos importantes de ficheros de sonido con compresión con pérdidas son:

- AAC (Advanced Audio Coding), sucesor de MP3. Forma parte de los estándares MPEG-2 y MPEG-4. Se utiliza en muchos de los últimos dispositivos móviles.
- Vorbis, libre de patentes (a diferencia de MP3 y AAC).
- WMA (Windows Media Audio), tecnología privativa que forma parte del «Windows Media Framework».

Hay mucha controversia sobre la calidad, pero los resultados prácticos son bastante parecidos. El factor de compresión está entre 10:1 y 12:1 para una tasa de bits de 128 kbps (aunque los códecs permiten aplicar la técnica VBR, *Variable Bit Rate*, con la que para la misma compresión se obtiene mayor calidad).

El factor de compresión no es el único parámetro importante, también hay que considerar la velocidad de ejecución del algoritmo (compresión y descompresión), la latencia en el caso de *streaming*, la calidad percibida subjetivamente, el soporte en hardware y en sistemas operativos, etc.

Ficheros de imágenes matriciales

Una imagen matricial *(raster image)* es una estructura de datos que representa una matriz de píxeles. Los tres parámetros importantes son el ancho y el alto en número de píxeles y el número de bits por píxel, **bpp**. A veces se le llama en general «*bitmap*», pero conviene distinguir entre:

- **bitmap** para imágenes en blanco y negro (1 bpp),
- **greymap** para imágenes en escala de grises (n bpp; el número de tonos es 2^n), y
- **pixmap** para imágenes en color (n bpp es la **profundidad de color**; el número de colores es 2^n)

Se puede ilustrar la idea con un gráfico genérico que muestre que el píxel (0,0) (el primero por la izquierda de la primera línea) se codifica con uno o varios bits, el (0,1) con los siguientes, etc., pero hagámoslo con un ejemplo sobre un formato concreto.

Uno de los formatos más sencillos es el **PBM** *(portable bitmap)*. Todo el contenido del fichero es texto plano. En la primera línea, el número mágico, «P1». En la segunda, el ancho, n, y el alto, m, en decimal. Y a continuación, m líneas. Cada línea tiene n caracteres ASCII que tienen que ser «0» (0x30) para «blanco» o «1» (0x31) para «negro», y pueden ir seguidos o separados por espacios (0x20). El símbolo «#» en cualquier línea indica que lo que sigue hasta el final de la misma es un comentario.

La figura 5.1 muestra el contenido de un fichero en formato PBM y el resultado de su visualización con un programa de gráficos. En realidad, este resultado está ampliado, porque si analiza usted el fichero verá que los trazos de las letras solo tienen dos píxeles, y las dimensiones de la imagen (32×10) son muy pequeñas.

```
P1    # Número mágico
32 10 # 32x10 píxeles
#   F   -    T    -   E    -   L
11111110 111111110 11111110 1100000
11111110 111111110 11111110 1100000
11000000 000110000 11000000 1100000
11000000 000110000 11000000 1100000
11111100 000110000 11111100 1100000
11111100 000110000 11111100 1100000
11000000 000110000 11000000 1100000
11000000 000110000 11000000 1100000
11000000 000110000 11111110 1111111
11000000 000110000 11111110 1111111
```

FTEL

Figura 5.1 FTEL.pbm.

Observe que los «1» y «0» no son bits, sino caracteres. Es decir, aunque sea un «bitmap», cada pixel ocupa realmente ocho bits.

Un formato similar para *pixmaps* es el **PPM** *(portable pixmap)*. En lugar de «1» y «0» contiene, para cada píxel, una tripla de números decimales, «R G B», que representan los niveles de rojo, verde y azul. En la cabecera, después del número mágico «P3» y de las dimensiones, en otra línea se indica el valor máximo, M, de la escala en la que se expresan estos niveles, de 0 a M. Normalmente M = 255, pero en el ejemplo mostrado en la figura 5.2 se ha utilizado M = 5.

```
P3    # Número mágico
32 10 # 32x10 pixels
5     # Máxima intensidad
#                     F                       -                     T                       -\
                      E                       -                     L
5 3 0 5 3 0 5 3 0 5 3 0 5 3 0 5 3 0 5 5 5  5 0 5 5 0 5 5 0 5 5 0 5 5 0 5 5 0 5 5 0 5 5 5 5\
0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 0 5 3 0 5 3 0 5 3 0 5 3 0 5 5 5  5 0 5 5 0 5 5 0 5 5 0 5 5 0 5 5 0 5 5 0 5 5 5 5\
0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  5 5 5 5 5 5 5 5 5 5 0 5 5 0 5 5 5 5 5 5 5 5 5 5\
0 5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  5 5 5 5 5 5 5 5 5 5 0 5 5 0 5 5 5 5 5 5 5 5 5 5\
0 5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 0 5 3 0 5 3 0 5 3 0 5 5 5 5 5 5  5 5 5 5 5 5 5 5 5 5 0 5 5 0 5 5 5 5 5 5 5 5 5 5\
0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 5 5 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 0 5 3 0 5 3 0 5 3 0 5 5 5 5 5 5  5 5 5 5 5 5 5 5 5 5 0 5 5 0 5 5 5 5 5 5 5 5 5 5\
0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 5 5 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 0 5 3 0 5 3 0 5 3 0 5 5 5 5 5 5  5 5 5 5 5 5 5 5 5 5 0 5 5 0 5 5 5 5 5 5 5 5 5 5\
0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 5 5 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  5 5 5 5 5 5 5 5 5 5 0 5 5 0 5 5 5 5 5 5 5 5 5 5\
0 5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5  5 0 0 5 0 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 3 0 5 3 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  5 5 5 5 5 5 5 5 5 5 0 5 5 0 5 5 5 5 5 5 5 5 5 5\
0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 5 5 5 5  5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0
5 3 0 5 3 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  5 5 5 5 5 5 5 5 5 5 0 5 5 0 5 5 5 5 5 5 5 5 5 5\
0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 5  5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0 5 0 0
```

Figura 5.2 FTEL.ppm.

Si analiza usted el contenido del fichero verá que los píxeles de la F están codificados con R = 5, G = 3, B = 0, lo que da un resultado naranja, los de la T con R = 5, G = 0, B = 5 (magenta), los de la E con R = 0, G = 5, B = 0 (verde) y los de la L con R = 5, G = 0, B = 0 (rojo). Los píxeles blancos, con R = 5, G = 5, B = 5. El símbolo «\» indica que la línea no acaba ahí, sino que continúa en la siguiente (en el fichero original no aparecen, cada una de las líneas de la imagen está representada en una sola línea de texto).

La única ventaja de estos dos formatos es que todo el contenido está codificado en texto plano, lo que los hace adecuados para la comunicación entre sistemas que solo están preparados para ese modo (como ocurría en los primeros tiempos del correo electrónico). Es obvio que se consigue una representación mucho más compacta codificando en binario. Los dos formatos son parte de una «familia» de formatos y programas de conversión de código abierto que se llama **Netpbm** y que incluye los resumidos en la tabla 5.1.

Número mágico	Tipo	Extensión	bpp
P1 (0x5031)	bitmap en ASCII	`.pbm`	8×1
P2 (0x5032)	graymap en ASCII	`.pgm`	entre 8×1 y 8×3
P3 (0x5033)	pixmap en ASCII	`.ppm`	entre $3 \times 8 \times 1$ y $3 \times 8 \times 3$
P4 (0x5034)	bitmap binario	`.pbm`	1
P5 (0x5035)	graymap binario	`.pgm`	8
P6 (0x5036)	pixmap binario	`.ppm`	24

Tabla 5.1 Formatos Netpbm.

Normalmente se aplican las versiones binarias. Por ejemplo, el programa `pngtopnm` convierte una imagen PNG a PBM si la imagen está en blanco y negro, a PGM si tiene grises y a PPM si tiene colores, pero siempre a los formatos binarios. Sin embargo, `pnmtopng` convierte de cualquiera de los formatos de la tabla 5.1 a PNG («PNM», o *«portable anymap»*, es el nombre común en Netpbm para PBM, PGM y PPM).

Netpbm es especialmente útil para operaciones de conversión entre otros formatos, como un formato intermedio. No se suele utilizar como formato final para la distribución de imágenes, porque hay otros, como PNG y GIF, más eficientes (ocupan menos espacio). El paquete contiene más de 220

programas de conversión y de manipulación. Está incluido en todas las distribuciones de las distintas variantes de Unix y tiene versiones para los demás sistemas operativos.

Los formatos de imágenes matriciales más conocidos son BMP, TIFF, GIF, PNG, y JPEG. De los tres últimos ya hemos comentado algo sobre los algoritmos de compresión que utilizan (apartados 4.4 y 4.5) y sus números mágicos (apartado 5.2). Algunos otros aspectos generales son:

- **BMP** *(Bitmap)* es el formato matricial de los sistemas Windows. Los detalles de los convenios de almacenamiento dependen de la profundidad de color, que se indica en la cabecera: 1, 2, 4, 8, 16 o 24 bpp. Cuando $n \leq 8$ bpp incluye una tabla que define una paleta de 2^n colores. En estos casos la imagen puede estar comprimida con RLE o Huffman, pero normalmente los ficheros BMP no están comprimidos.

- **TIFF** *(Tagged Image File Format)* es el más conocido en el mundo profesional fotográfico. Se utiliza sin compresión, o con compresión sin pérdidas (LZW), para archivos de imágenes y para aplicaciones que requieren alta resolución y fidelidad, como las imágenes médicas.

- **GIF** *(Graphics Interchange Format)* es el formato más extendido en la web para imágenes sencillas. Su poca profundidad de color (8 bpp: 256 colores) lo hace poco adecuado para fotografías. Utiliza para la compresión LZW, que tenía el problema de estar patentado, pero la patente expiró en 2004. En un mismo fichero se pueden incluir varias imágenes con metadatos para presentarlas secuencialmente, lo que permite hacer animaciones sencillas («GIFs animados»).

- **PNG** *(Portable Network Graphics)* surgió como alternativa a GIF por el problema de la patente (utiliza Deflate en lugar de LZW, apartado 4.4) y tiene más opciones, como la posibilidad de añadir un nivel de transparencia a los píxeles («canal alpha»). Normalmente, el factor de compresión es algo mejor que con GIF, pero no siempre es así (depende del programa que genera las imágenes y de la imagen misma). No permite animaciones, aunque hay dos extensiones, MNG *(Multiple-image Network Graphics)* y APNG *(Animated Portable Network Graphics)*, poco utilizadas.

- **JPEG** *(Joint Photographic Experts Group)* es el formato más utilizado para la distribución de fotografías, por su flexibilidad y la eficiencia de los algoritmos de compresión (apartado 4.5). Las herramientas que generan o manipulan los ficheros con este formato suelen ofrecer un parámetro global de calidad, Q, cuyo valor puede elegirse entre 0 y 100. Para Q = 0 se obtiene la máxima compresión (mínimo tamaño, mínima calidad), y para Q = 100 la mínima (máximo tamaño, máxima calidad).

Ficheros de imágenes vectoriales

La representación «vectorial» de las imágenes es un enfoque totalmente distinto al de la representación «matricial». En lugar de «ver» la imagen como una matriz de píxeles, se la *describe* como un conjunto de objetos primitivos (líneas, círculos, etc.) definidos matemáticamente en un lenguaje.

Para entenderlo, veamos un ejemplo concreto en el lenguaje SVG *(Scalable Vector Graphics)*, que es un estándar del W3C (Consorcio WWW). En la figura 5.3 puede usted ver el código en SVG y el resultado visual de una imagen sencilla.

Aun sin saber nada del lenguaje (que está basado en XML, un metalenguaje que estudiaremos en el Tema 3), es fácil reconocer la correspondencia entre las sentencias y la imagen. Después de la cabecera (las cinco primeras líneas), aparecen tres declaraciones de rectángulos con sus propiedades (posición, dimensiones y color).

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg"
     width="100%" height="100%">
<rect x="0" y="0" width="240" height="50"
      stroke="red" stroke-width="1" fill="red" />
<rect x="0" y="50" width="240" height="70"
      stroke="yellow" stroke-width="1" fill="yellow" />
<rect x="0" y="120" width="240" height="50"
      stroke="red" stroke-width="1" fill="red" />
</svg>
```

Figura 5.3 Bandera.svg.

Con el programa `convert` del paquete «ImageMagick» se pueden generar a partir de ese fichero otros con la misma imagen en otros formatos, y así comparar los tamaños. Estos son los resultados para GIF y PNG con resolución de 241×171:

```
$ls -l --sort=size bandera.*
-rw-r--r-- 1 gfer gfer 836 jul 21 20:26 bandera.gif
-rw-r--r-- 1 gfer gfer 663 jul 21 20:37 bandera.png
-rw-r--r-- 1 gfer gfer 486 jul 21 20:24 bandera.svg
```

Para imágenes no tan simples el código es, por supuesto, más largo. La figura 5.4 procede de un fichero SVG que ocupa 16 KiB. Las conversiones a GIF y a PNG con resolución 341×378 y 16 bpp dan lugar a ficheros de 15 KiB y 60 KiB, respectivamente. Como se ve, PNG no siempre ocupa menos que GIF: depende de la imagen, los colores, etc. En el fichero SVG unos 2 KiB son de la cabecera con metadatos sobre el autor, la licencia (dominio público), etc.

Figura 5.4. Beastie (Logo de freeBSD).

El fichero SVG completo es demasiado grande para incluirlo aquí; en la figura 5.5 se reproduce una pequeña parte: la que dibuja el cuerno derecho.

```
 ...
 <path d="M125.64,39.465
    c-3.15-3.979-5.49-6.699-5.22-11.58
    c1.3-11.46,21.03-20.449,18.58-26.39
    c-1.77-4.29-18.9,7.28-26.54,13.21
    c-21.54,16.73-39.66,35.03-23.63,62.37
    C92.14,66.315,111.47,37.095,125.64,39.465z"
    fill="#CB1009" stroke="#000000" stroke-width="0.5"/>
...
```

Figura 5.5 El cuerno derecho del demonio.

La sentencia «`<path d=.../>`» especifica las coordenadas de un punto de origen (125.64,39.465) seguidas de las coordenadas relativas de otros puntos que se unen cada uno al anterior mediante curvas de Bézier cúbicas («`c`») y termina con las mismas coordenadas del punto de origen, indicando el color de relleno y el grosor y el color del contorno.

Generalmente, el tamaño del fichero con la imagen vectorial es similar o menor que el del matricial, como se puede apreciar en los datos de los ejemplos anteriores. Además, al tratarse de texto plano se puede comprimir bastante. Veamos los resultados antes y después de aplicarles `gzip`:

```
$ls -l --sort=size freebsd-daemon.*
-rw-r--r-- 1 gfer gfer 61242 jul 26 10:57 freebsd-daemon.png
-rw-r--r-- 1 gfer gfer 16531 jul 26 00:09 freebsd-daemon.svg
-rw-r--r-- 1 gfer gfer 15780 jul 26 10:57 freebsd-daemon.gif
$gzip freebsd-daemon.*
$ls -l --sort=size freebsd-daemon.*
-rw-r--r-- 1 gfer gfer 55847 jul 26 10:57 freebsd-daemon.png.gz
-rw-r--r-- 1 gfer gfer 15239 jul 26 10:57 freebsd-daemon.gif.gz
-rw-r--r-- 1 gfer gfer  7055 jul 26 00:09 freebsd-daemon.svg.gz
```

Los ficheros GIF y PNG apenas se han reducido (los formatos originales ya están comprimidos), pero para el SVG ha resultado un factor de compresión aproximado de 2,34:1 ($\approx 43\,\%$).

La representación vectorial tiene otra ventaja: es independiente de la resolución. Una imagen matricial tiene unos números fijos de píxeles horizontales y verticales, y no se puede ampliar arbitrariamente sin perder calidad (sin «pixelarse»). La imagen vectorial se puede ampliar todo lo que se necesite: la calidad solo está limitada por el hardware en el que se presenta.

Por otra parte, la representación vectorial solo es aplicable a los dibujos que pueden describirse mediante primitivas geométricas. Para imágenes fotográficas es totalmente inadecuada.

La conclusión es que depende de la aplicación el que sea mejor trabajar con formatos y herramientas matriciales o vectoriales, o una combinación de ambos. Observe la palabra «herramientas». Como el formato vectorial es texto plano, puede componerse una imagen con un editor de textos normal. Así se ha hecho la imagen de la figura 5.3. Para dibujos no triviales sería demasiado laborioso, y es imprescindible utilizar un programa (herramienta). La imagen de la figura 5.4 (procedente del proyecto freeBSD) está compuesta con «Adobe Illustrator». Hay programas de dibujo de código abierto que tienen SVG como lenguaje «nativo», como Inkscape (`http://www.inkscape.org/`), o que pueden exportar los dibujos a SVG, como xfig (`http://www.xfig.org`).

Ficheros de vídeo

Algunos formatos para ficheros de vídeo se limitan a hacer una compresión intrafotograma (apartado 4.5), con la que consiguen factores de compresión comprendidos entre 5:1 y 10:1. Por ejemplo:

- **DV** *(Digital Video)* se propuso en 1996 por un consorcio de 60 fabricantes para los mercados industrial y doméstico, y para almacenamiento en cinta. Se ha ido modificando para otros medios y para videocámaras, con diversas variantes según los fabricantes.

- **MJPEG** *(Motion JPEG)* consiste en comprimir cada fotograma en JPEG. No es un estándar, y no hay documentos de especificaciones. Se utiliza en aplicaciones de edición: al no haber compresión interfotograma se pueden manipular las imágenes individualmente con herramientas de JPEG.

Los formatos más conocidos para la transmisión de vídeo (y audio) y para su almacenamiento en ficheros son los que especifican las normas del MPEG (*Moving Picture Experts Group*, un grupo de trabajo de ISO e IEC), y sus derivados. Hay tres estándares básicos:

- **MPEG-1**. El vídeo se comprime con los procedimientos que mencionamos al final del apartado 4.5 para una calidad VHS con un factor de compresión 26:1 y una tasa de bits de 1,5 Mbps, y se utiliza, por ejemplo, en Video CD (VCD), un formato para almacenar vídeo en CD de audio. La parte de audio es la popularmente conocida como MP3.

- **MPEG-2** introduce algunas mejoras sobre MPEG-1 en vídeo, y, sobre todo, en audio, con AAC, ya comentado antes en los ficheros de sonidos. Es la base de otros estándares, como DVB-T *(Digital Video Broadcasting - Terrestrial)* (el utilizado para lo que en España se llama TDT) y DVD-Video.

- **MPEG-4**, que en principio estaba orientado a comunicaciones con ancho de banda limitado, se ha ido extendiendo a todo tipo de aplicaciones con anchos de banda desde pocos kbps a decenas de Mbps. Aparte de mejorar los algoritmos de compresión de MPEG-2 (se obtiene la misma calidad con la mitad de tasa de bits), incluye otras características, como **VRML** *(Virtual Reality Modeling Language)* que permite insertar objetos interactivos en 3D. La parte de vídeo, H.264/MPEG-4 Part 10, se conoce como **H.264** o **AVC** *(Advanced Video Coding)* y es actualmente uno de los formatos más utilizados para compresión, almacenamiento y distribución de vídeo de alta definición, por ejemplo, en HDTV, en los discos Blu-ray y en la web.

Hay varios formatos derivados de o relacionados con los estándares MPEG:

- **WMV** *(Windows Media Video)*, inicialmente basado en MPEG-4, evolucionó a un formato propietario que ha sido estandarizado por SMPTE *(Society of Motion Picture and Television Engineers)* con el nombre «VC-1». Se usa también en Blue-ray y en la Xbox 360.

- **DivX**, que tiene una historia tortuosa: su origen fue una copia (ilegal) de WMV llamada «DivX ;-)»; luego se comenzó a reelaborar completamente como software libre («Proyecto Mayo»), pero los líderes del proyecto terminaron creando una empresa, DivX, Inc., que comercializa códecs privativos. Paralelamente, algunos colaboradores del proyecto continuaron con la versión libre, que actualmente se llama «Xvid».

- **F4V**, que está basado en H.264, es el formato más reciente del muy extendido «Flash Player» (el formato FLV está basado en un estándar anterior, el H.263).

- **Theora**, desarrollado por la Fundación Xiph.org, es, a diferencia de los anteriores, libre: sus especificaciones son abiertas y los códecs son de código abierto. Desarrollado por Xiph.org Foundation. Es comparable en tecnología y eficiencia a MPEG-4 y WMV.

- **VP8**, creado por una empresa (On 2 Technologies) que fue comprada por Google y actualmente lo distribuye como sofware libre.

Contenedores multimedia

Un formato contenedor, o «envoltorio» *(wrapper)* es un conjunto de convenios para definir cómo se integran distintos tipos de datos y de metadatos en un solo fichero. Es especialmente útil en multimedia, para distribuir y almacenar conjuntamente ficheros de naturaleza diferente pero relacionados: varios canales de sonido, vídeos, subtítulos, menús de vídeo interactivos, etc.

Algunos de los formatos de vídeo mencionados antes, como FLV y F4V, son en realidad contenedores. En la tabla 5.2 se resumen algunas características de otros, incluyendo también las extensiones que suelen ponerse en los nombres de los ficheros.

Contenedor	Origen	Comentarios	Ext.
QuickTime	Apple	Uno de los más antiguos. Puede contener diversos formatos de audio y de vídeo. MP4 está basado en él.	.mov
MP4 (MPEG-4 Part 14)	MPEG	Contenedor oficial para los formatos de audio (AAC) y vídeo (AVC) de MPEG-4, pero admite otros formatos.	.mp4, .m4a, .m4v
3GP (3rd Generation Partnership Project)	3GPP	Adaptación de MP4 para servicios multimedia UMTS. Vídeo AVC y varios formatos de audio.	.3gp
AVI (Audio Video Interleave)	Microsoft	Contenedor habitual de Windows desde 1992. Obsolescente.	.avi
ASF (Advanced Streaming Format)	Microsoft	Reemplazo de AVI.	.asf
Matroska	matroska.org	Formato universal para contener todo tipo de contenido multimedia. Especificaciones abiertas e implementaciones libres.	.mkv
Ogg	Xiph.org	Contenedor para el códec de audio Vorbis y el de vídeo Theora. Todos ellos, software libre.	.ogg
WebM	Google	Basado en Matroska, con códecs Vorbis para audio y VP8 para vídeo, también es de código abierto. Orientado a la web (HTML5).	.webm

Tabla 5.2 Algunos contenedores de multimedia

Multimedia en la web

Habiendo mencionado varios formatos de audio, vídeo y contenedores, es interesante comentar, como nota marginal, una controversia actual sobre la distribución de contenido multimedia en la web. (Es «marginal» porque es efímera: en unos años la situación habrá cambiado).

El formato de vídeo más extendido en las páginas web es H.264/MPEG-4 AVC, que está sujeto a patentes. Por otra parte, la última versión del lenguaje HTML, aún en desarrollo, es HTML5, que incluye elementos para insertar directamente objetos multimedia sin recurrir a complementos *(plugins)* privativos (como «Flash Player»). Las especificaciones iniciales recomendaban que todos los «agentes de usuario» (navegadores) deberían incluir soporte para un formato de vídeo libre de patentes, concretamente, Theora para vídeo y Vorbis para audio, con Ogg como contenedor. Varias empresas se opusieron con el argumento de que esos formatos vulneran patentes, y en diciembre de 2007 se eliminó ese requisito. Más confusión se ha introducido recientemente con la introducción por Google del formato VP8 y el contenedor WebM y su decisión de abandonar el soporte para H.264 en Chrome, puesto que las mismas empresas le acusan de violar patentes.

Actualmente (2012) los navegadores Internet Explorer y Safari tienen soporte «nativo» (es decir, sin necesidad de instalar complementos) para H.264, pero no para Ogg ni WebM, mientras que en Firefox, Chromium (versión libre de Chrome) y Opera es lo contrario. Chrome, de momento, como también los navegadores incluidos en Android, aceptan los tres.

En cualquier caso, esta controversia no tiene prácticamente ninguna repercusión para el usuario final, puesto que los complementos, aunque privativos y sujetos a patentes, se distribuyen gratuitamente.

5.5. Ficheros ejecutables binarios

Los ficheros ejecutables contienen programas, es decir, órdenes, o instrucciones, para ser ejecutadas por un procesador, a diferencia de los ficheros estrictamente de datos, cuyo contenido «alimenta» a los programas.

Ahora bien, como veremos en el Tema 3, hay varios tipos de lenguajes de programación. En algunos, esas «órdenes o instrucciones» se escriben en un formato de texto plano, para que un programa llamado **intérprete** las vaya leyendo *una tras otra* y generando, para cada una, las acciones adecuadas. Un programa de ese tipo es el intérprete de órdenes (o «shell») con el que hemos practicado en el Tema 1. Un fichero puede contener un programa formado una sucesión de tales órdenes (lo que se suele llamar un *«script»*), y podemos decir de él que es «ejecutable».

Pero cuando decimos «ejecutables binarios» nos referimos a otra cosa. El único lenguaje que «entiende» el procesador es un lenguaje binario, el **lenguaje de máquina**. Cada procesador tiene el suyo, definido por las operaciones básicas o **instrucciones** que, expresadas en binario, es capaz de ejecutar. Hay lenguajes de programación diseñados para escribir programas inteligibles para las personas que han de ser *traducidos* al lenguaje de máquina del procesador. Los programas escritos en estos lenguajes se llaman **programas fuente** y se guardan en ficheros de texto plano. A diferencia del intérprete, el programa **traductor** lee un programa fuente y *de una sola vez* genera el equivalente en el lenguaje de máquina del procesador. Este programa generado se llama **programa objeto**, o **código objeto**. Cuatro observaciones:

- «Binario» se refiere a «ejecutable», no a «fichero», aunque a veces se diga también «ficheros binarios» para referirse a los que no son de texto plano. Esta expresión es algo falaz, porque parece implicar que los ficheros de texto no son binarios, y, sin embargo, aunque su contenido se interprete como una sucesión de codificaciones de caracteres, siempre será binario.

- La distinción entre «programa» y «datos» es relativa: un programa fuente es, obviamente, un programa, pero para el programa traductor son datos de entrada.

- Hay una diferencia esencial entre este tipo de fichero (ejecutable binario) y los anteriores (documentos, multimedia, etc.), y es que su formato no puede independizarse del entorno de ejecución: está íntimamente ligado al procesador y al sistema operativo en los que va a ejecutarse.

El traductor guarda el código objeto en un fichero para que, en cualquier momento posterior, el sistema lo cargue en la memoria y se pueda ejecutar[2]. No intente usted leer (con `cat`, `less` etc.) un fichero de este tipo (ni de ningún tipo que no sea texto plano), porque no verá nada más que símbolos raros (y eventualmente perderá el control del terminal, al aparecer accidentalmente la codificación de algún carácter de control). Sí puede «verlo» con un programa como «`hd`» (o «`hexdump`»), que muestra en hexadecimal los contenidos binarios. Pruebe a hacerlo con cualquier fichero del directorio `/bin` de un sistema Unix. Por ejemplo, con la orden «`hd /bin/ls | less`». Verá cómo al principio aparece el número mágico (codificaciones ASCII de «.ELF»).

Como cada procesador tiene su lenguaje, el traductor tiene que estar adaptado al procesador concreto para el que traduce. El binario resultante, además del número mágico, tiene una cabecera que informa de qué procesador se trata, y un determinado formato. Y como los programas de aplicación hacen un uso sistemático de las llamadas al sistema (Tema 1) para acceder a los recursos de hardware,

[2]Como veremos en el Tema 3, lo que genera el traductor no es directamente ejecutable, tiene que pasar por otro proceso, llamado «montaje».

ese formato está especialmente ligado al sistema operativo. Cada uno tiene sus convenios. No tiene sentido entrar aquí en los detalles, pero sí citar los tres formatos más comunes actualmente:

- **PE** (Portable Executable) en los sistemas Windows
- **ELF** (Executable and Linkable Format) en los sistemas Unix.
- **Mach-O** (Mach Object) en los sistemas Darwin y Mac OS X.

Binarios gordos y *blobs*

Cuando usted instala un programa nuevo en su ordenador tiene dos opciones:

- La del «experto»: si dispone del programa fuente, lo traduce (normalmente, con un compilador del lenguaje en el que está escrito el programa) y genera un código objeto adaptado a su ordenador y a su sistema operativo. Pero esta opción no siempre es posible, ni siendo experto, porque muchos programas son privativos, y no se dispone del código fuente.
- La «normal»: ejecuta un programa instalador proporcionado por el distribuidor del programa, que automáticamente extrae el código objeto ya adaptado a su sistema y lo guarda en un fichero, añadiendo datos al SGF para que pueda localizarlo.

La segunda opción implica que el distribuidor debe hacer versiones para los distintos procesadores y sistemas operativos. Un enfoque para reducir esta variedad es que el código objeto incluya versiones para varios procesadores. Naturalmente, tiene un tamaño mayor que el generado para un procesador concreto, y de ahí el nombre de *«fat binary»*, en el que «fat» tiene su significado literal en inglés (nada que ver con la tabla de asignación de ficheros, apartado 5.1). Dos ejemplos de este enfoque son:

- El formato *«Universal Binary»* de Apple, para programas que pueden ejecutarse tanto en los microprocesadores PowerPC como en los Intel (Apple cambió de unos a otros en 2005).
- El formato *«FatELF»* de Linux y otros Unix, que es una extensión del formato ELF.

«**Blob**» es un acrónimo de *binary large object* y en principio es un término ligado a las bases de datos. El *Free On-Line Dictionary Of Computing* (`http://foldoc.org`) lo define así:

> *Un gran bloque de datos almacenado en una base de datos, como un fichero de sonido o de imagen. Un BLOB no tiene una estructura que pueda ser interpretada por el sistema de gestión de la base de datos, éste lo conoce únicamente por su tamaño y localización.*

Pero el término se utiliza también con un sentido más general, para referirse a código binario cuyo programa fuente no está disponible. Este uso está ligado a la cultura del software libre. Por ejemplo, los núcleos de sistemas operativos como Linux, FreeBSD, NetBSD, etc., son «libres» (o de «código abierto») lo que significa, entre otras cosas, que todos los programas fuente se distribuyen libremente. El problema es que algunos fabricantes de controladores de periféricos (tarjetas gráficas, de red, etc.) facilitan el código binario de los gestores (*drivers*, Tema 1), pero no los fuentes ni la información técnica necesaria para poder programar esos fuentes. En algunas distribuciones de estos sistemas operativos se incluyen esos gestores en binario, y se dice de ellos (con un cierto toque despectivo) que son «blobs».

Apéndice: Sistemas de numeración posicionales

Bases de numeración

El sistema de numeración decimal utiliza diez cifras o dígitos y un convenio que consiste en atribuir a cada dígito un valor que depende de su posición en el conjunto de dígitos que representa al número. Así, por ejemplo:
$3.417 = 3 \times 10^3 + 4 \times 10^2 + 1 \times 10^1 + 7 \times 10^0$;
$80,51 = 8 \times 10^1 + 0 \times 10^0 + 5 \times 10^{-1} + 1 \times 10^{-2}$

Decimos que este sistema tiene base 10. La base, k, es el cardinal del conjunto de símbolos (o *alfabeto*) utilizados para la representación del número. Si $k > 10$, entonces los diez dígitos decimales no son suficientes; es convenio generalizado el utilizar las letras sucesivas, a partir de la A. Así, para $k = 16$ (sistema hexadecimal), el alfabeto es:

$$\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\},$$

y el número AF1C, por ejemplo, tendrá la siguiente representación decimal:
$10 \times 16^3 + 15 \times 16^2 + 1 \times 16 + 12 = 44.828$

En general, si $a_n a_{n-1} \ldots a_1 a_0, a_{-1} \ldots a_{-m}$ está escrito en base k, entonces su valor en base 10 puede calcularse así:
$a_n \times k^n + a_{n-1} \times k^{n-1} + \ldots + a_1 \times k + a_0 + a_{-1} \times k^{-1} + \ldots + a_{-m} \times k^{-m}$

Cambios de base

El problema general, considerando de momento solo números enteros, consiste en que, dado un número expresado en base k:

$$N_{(k)} = a_p a_{p-1} \ldots a_1 a_0,$$

se desea hallar su representación en base h:

$$N_{(h)} = b_q b_{q-1} \ldots b_1 b_0$$

La forma más sencilla de resolverlo (porque estamos acostumbrados a operar en decimal) es pasar por el intermedio de la base 10. Así, de $N_{(k)}$ calcularemos $N_{(10)}$ mediante las potencias sucesivas. Para obtener luego $N_{(h)}$ a partir de $N_{(10)}$ tendremos en cuenta que:
$N_{(10)} = b_0 + b_1 \times h + b_2 \times h^2 + \ldots + b_{q-1} \times h^{q-1} + b_q \times h^q = b_0 + N1_{(10)} \times h$
con
$N1_{(10)} = b_1 + h \times N2_{(10)}$;
$N2_{(10)} = b_2 + h \times N3_{(10)}$, etc.

De aquí,

$b_0 = N_{(10)}$ módulo $h_{(10)}$ (resto de la división $N_{(10)}/h$)

$b_1 = N1_{(10)}$ módulo $h_{(10)}$, etc.

Es decir, el procedimiento consiste en dividir sucesivamente el número y los cocientes que vayan resultando por la base a la que queremos pasar, y los dígitos de la representación en esta base serán los restos que vayan resultando.

Por ejemplo, para representar en base 12 (alfabeto: {0,1,2,3,4,5,6,7,8,9,A,B}) el número $4432_{(5)}$ escrito en base 5 (alfabeto: {0,1,2,3,4}) operaremos así:

$N_{(10)} = 4 \times 5^3 + 4 \times 5^2 + 3 \times 5 + 2 \times 5^0 = 617$

$617 \div 12 = 51 + 5/12; b_0 = 5;$

$51 \div 12 = 4 + 3/12; b_1 = 3;$

$4 \div 12 = 0 + 4/12; b_2 = 4;$

Luego $4432_{(5)} = 435_{(12)}$

Otro ejemplo. Conversión de $4664_{(7)}$ a base 12:

$N_{(10)} = 4 \times 7^3 + 6 \times 7^2 + 6 \times 7 + 4 = 1712$

$1712 \div 12 = 142 + 8/12; b_0 = 8;$

$142 \div 12 = 11 + 10/12; b_1 = 10_{(10)} = \mathrm{A}_{(12)};$

$11 \div 12 = 0 + 11/12; b_2 = 11_{(10)} = \mathrm{B}_{(12)};$

Luego $4664_{(7)} = \mathrm{BA8}_{(12)}$

El paso intermedio por el sistema decimal no sería necesario si hiciésemos todas las divisiones en la base k de partida, pero resultaría más incómodo, al no estar acostumbrados a operar en otra base que la decimal.

El procedimiento para la parte fraccionaria es parecido, pero con multiplicaciones sucesivas en lugar de divisiones. En efecto, sea un número fraccionario M, entre 0 y 1, expresado en la base k:

$M_{(k)} = 0, a_{-1}a_{-2}\ldots$

y queremos hallar su representación en base h:

$M_{(h)} = 0, b_{-1}b_{-2}\ldots$

Primero lo pasamos a base 10, y luego tenemos en cuenta que

$M_{(10)} = b_{-1} \times h^{-1} + b_{-2} \times h^{-2} + \ldots$

Para hallar b_{-1} haremos:

$h_{(10)} \times M_{(10)} = b_{-1} + M'_{(10)}$

Por tanto, b_{-1} = (parte entera de $h_{(10)} \times M_{(10)}$).

Análogamente,

b_{-2} = (parte entera de $h_{(10)} \times M_{(10)}$),

y así sucesivamente.

Si queremos, por ejemplo, escribir en base 12 el número $M = 0,01_{(5)}$ operaremos de este modo:

$M_{(10)} = 1 \times 5^{-2} = 1/25 = 0,04$

$12 \times 0,04 = 0,48 = 0 + 0,48; b_{-1} = 0$

$12 \times 0,48 = 5,76 = 5 + 0,76; b_{-2} = 5$

$12 \times 0,76 = 9,12 = 9 + 0,12; b_{-3} = 9$

$12 \times 0,12 = 1,44 = 1 + 0,44; b_{-4} = 1$

$12 \times 0,44 = 5,28 = 5 + 0,28; b_{-5} = 5$

$12 \times 0,28 = 3,36 = 3 + 0,36; b_{-6} = 3$

$12 \times 0,36 = 4,32 = 4 + 0,32; b_{-7} = 4$

Resultando: $0,01_{(5)} = 0,0591534\ldots_{(12)}$.

Bases octal y hexadecimal

La conversión de una base a otra resulta muy fácil si una de ellas es potencia de la otra, pues en ese caso basta con reagrupar los dígitos.

Sea por ejemplo un número expresado en binario,

$N_{(2)} = a_p a_{p-1} a_{p-2} \ldots a_1 a_0$

del que queremos hallar su representación en base 8, o **representación octal** ($2^3 = 8$):

$N_{(8)} = b_q b_{q-1} b_{q-2} \ldots b_1 b_0$

Si hacemos la siguiente agrupación:

$N_{(10)} = a_0 \times 2^0 + a_1 \times 2^1 + a_2 \times 2^2 +$
$a_3 \times 2^3 + a_4 \times 2^4 + \ldots =$
$= (a_0 + 2 \times a_1 + 4 \times a_2) +$
$(a_3 + 2 \times a_4 + 4 \times a_5) \times 2^3 +$
$(a_6 + 2 \times a_7 + 4 \times a_8) \times 2^6 + \ldots$

y llamamos

$b_0 = (a_0 + 2 \times a_1 + 4 \times a_2)_{(10)} = (a_2 a_1 a_0)_{(2)}$
$b_1 = (a_3 + 2 \times a_4 + 4 \times a_5)_{(10)} = (a_5 a_4 a_3)_{(2)}$
$b_2 = (a_6 + 2 \times a_7 + 4 \times a_8)_{(10)} = (a_8 a_7 a_6)_{(2)}$

etc., podemos escribir:

$N_{(10)} = b_0 + b_1 \times 2^3 + b_2 \times 2^6 + \ldots$
$= b_0 \times 8^0 + b_1 \times 8^1 + b_2 \times 8^2 + \ldots$

Vemos así que los dígitos de la representación octal de un número binario pueden obtenerse directamente, agrupando los bits de tres en tres, y lo mismo ocurre con los números fraccionarios. He aquí un par de ejemplos:

$$\underbrace{100}_{4}\underbrace{110}_{6} = 46_{(8)}$$

$$\underbrace{11}_{3}\underbrace{111}_{7}\underbrace{000}_{0}, \underbrace{010}_{2}\underbrace{001}_{1}\underbrace{1}_{4} = 370,214_{(8)}$$

Por esta razón, la base octal se puede utilizar como una manera compacta de escribir números binarios, o cualquiera otra información codificada en binario.

Como la unidad por encima del bit es el byte, generalmente es más cómodo utilizar la base 16, o **hexadecimal**. Un desarrollo similar al anterior nos conduciría a que la equivalencia se obtiene agrupando los bits de cuatro en cuatro. Así, los dos números anteriores se escribirían del siguiente modo en hexadecimal:

$$\underbrace{10}_{2}\underbrace{0110}_{6} = 26_{(16)}$$

$$\underbrace{1111}_{F}\underbrace{1000}_{8}, \underbrace{0100}_{4}\underbrace{011}_{6} = \mathrm{F8},46_{(16)}$$

La representación hexadecimal es más compacta que la octal; tiene el inconveniente de necesitar, además de los diez dígitos decimales, seis «dígitos» más, A, B, C, D, E, F, cuyos valores decimales respectivos son 10, 11, 12, 13, 14 y 15. Con un poco de práctica, sin embargo, no es difícil interpretar expresiones como «efemil acientos benta y ce» (FABC). Más laborioso, absolutamente inútil y solo interesante para adictos a pasatiempos extravagantes, sería aprender a operar aritméticamente con esta base.

Para eludir el uso de subíndices se pueden seguir distintas notaciones. La más común (derivada de Unix y el lenguaje C) es utilizar el prefijo «0x» para indicar hexadecimal, «0b» para binario y «0» para octal. Si el primer carácter es un dígito entre 1 y 9 se entiende que es decimal. He aquí dos ejemplos de interpretación decimal de un número binario pasando por la representación hexadecimal:

$$0b\underbrace{10}_{2}\underbrace{1001}_{9}\underbrace{1110}_{E}\underbrace{1111}_{F} = 0x29EF = 2 \times 16^3 + 9 \times 16^2 + 14 \times 16 + 15 = 10735$$

$$0b\underbrace{111}_{7}\underbrace{1010}_{A}\underbrace{0000}_{0}\underbrace{1100}_{C}\underbrace{1101}_{D} = 0x7A0CD = 7 \times 16^4 + 10 \times 16^3 + 12 \times 16^1 + 13 = 499917$$

Bibliografía

Para complementar o ampliar el contenido de este Tema hay numerosas fuentes que puede usted localizar buscando en la web. Para reforzar los conceptos con ejercicios, algunos libros recomendables son:

- A. Prieto, A. Lloris y J.C. Torres: *Introducción a la informática,* 4ª *ed.* McGraw–Hill, Madrid, 2006.

  La representación de datos se trata en el Capítulo 4, con numerosos ejercicios propuestos.

- A. Prieto y B. Prieto: *Conceptos de Informática*. Serie Schaum, McGraw–Hill, Madrid, 2005.

  Es un texto complementario del anterior, que incluye gran cantidad de ejercicios (487), algunos resueltos (219), y 560 ejercicios de tipo test.

- J. Glenn Brookshear: *Computer Science. An Overview, 11th. ed.* Addison-Wesley, 2011.
  Traducido al español con el título *Introducción a la computación*. Pearson Educación, Madrid, 2012.

  El Capítulo 1 trata, esencialmente, los mismos conceptos que hemos estudiado en este Tema. Incluye preguntas y ejercicios resueltos y muchos problemas propuestos.

- N. Dale y J. Lewis: *Computer Science Illuminated, 4th. ed.* Jones and Barlett, 2011.

  En el capítulo 2 introduce los sistemas de numeración, y en el 3 la representación, incluyendo multimedia, pero a un nivel más superficial. También tiene muchos ejercicios propuestos.